Combinational logic

Digital Systems – Lecture 17

Department of
COMPUTER
SCIENCE

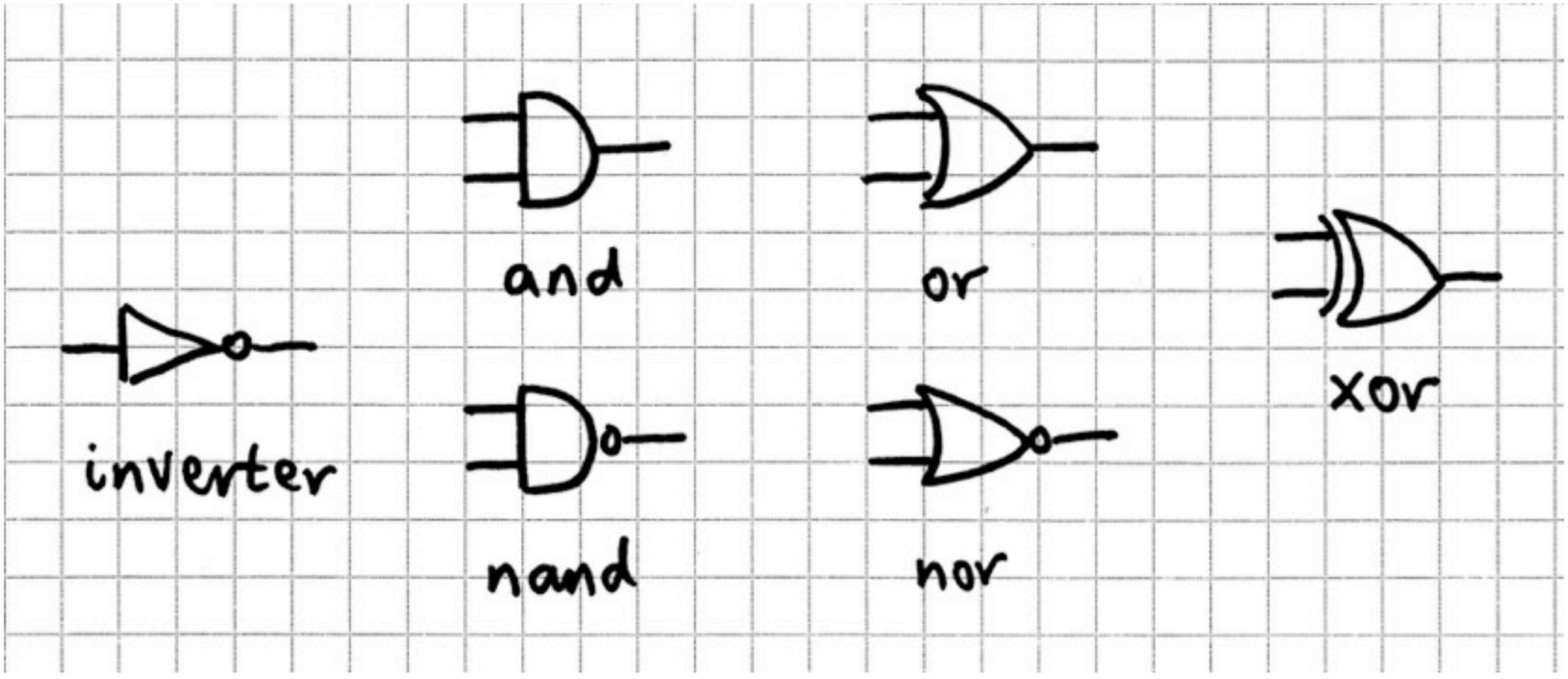
In this part

- Use transistors to build logic gates (L18)
- Use logic gates and latches to build functional units (L17, 19, 20)
- Use functional units to design a datapath that can execute Thumb code (L21, 22)

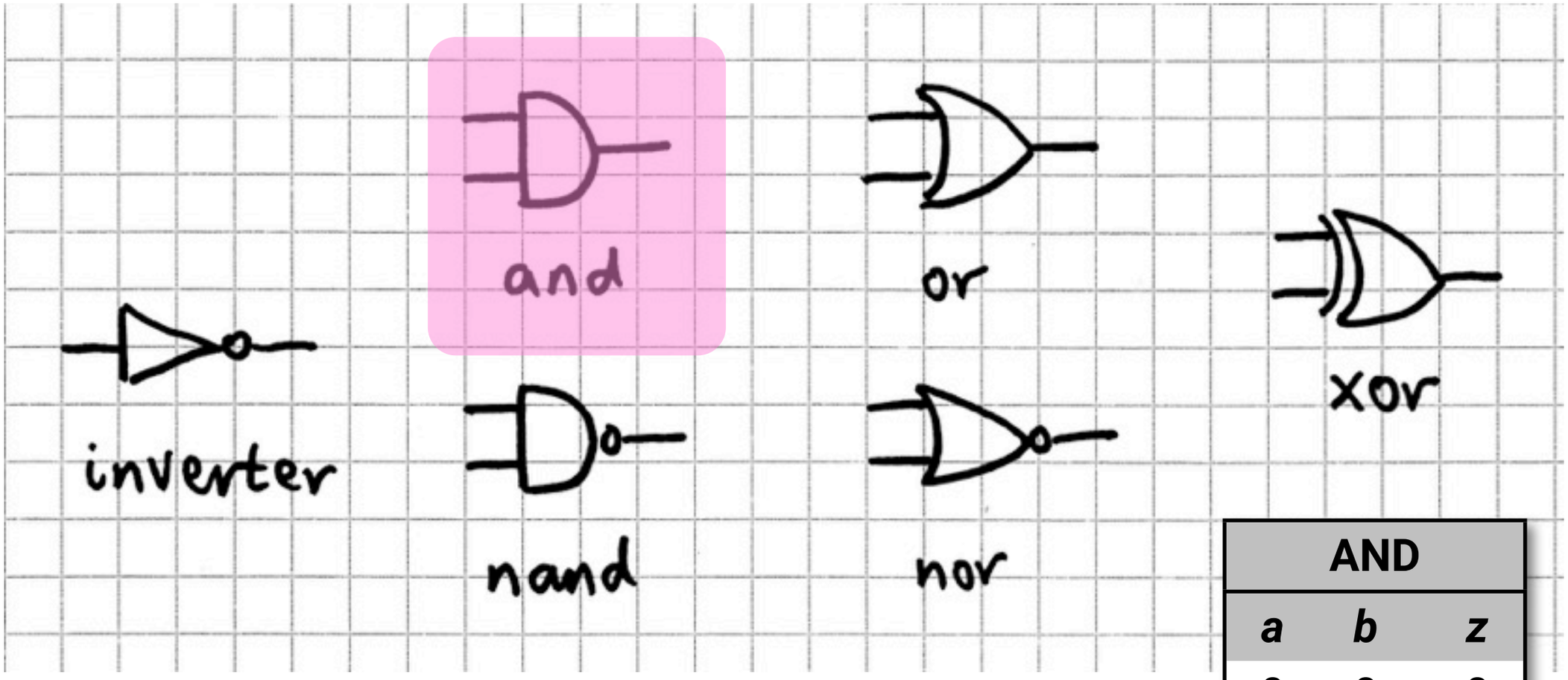
In this lecture

- *Combinational circuits* implement Boolean functions using logic gates.
- Any Boolean function can be implemented by combining a few types of simple gates.
- The speed of a combinational circuit is measured by the maximum *propagation delay* from input to output.

[17.1] Logic gates

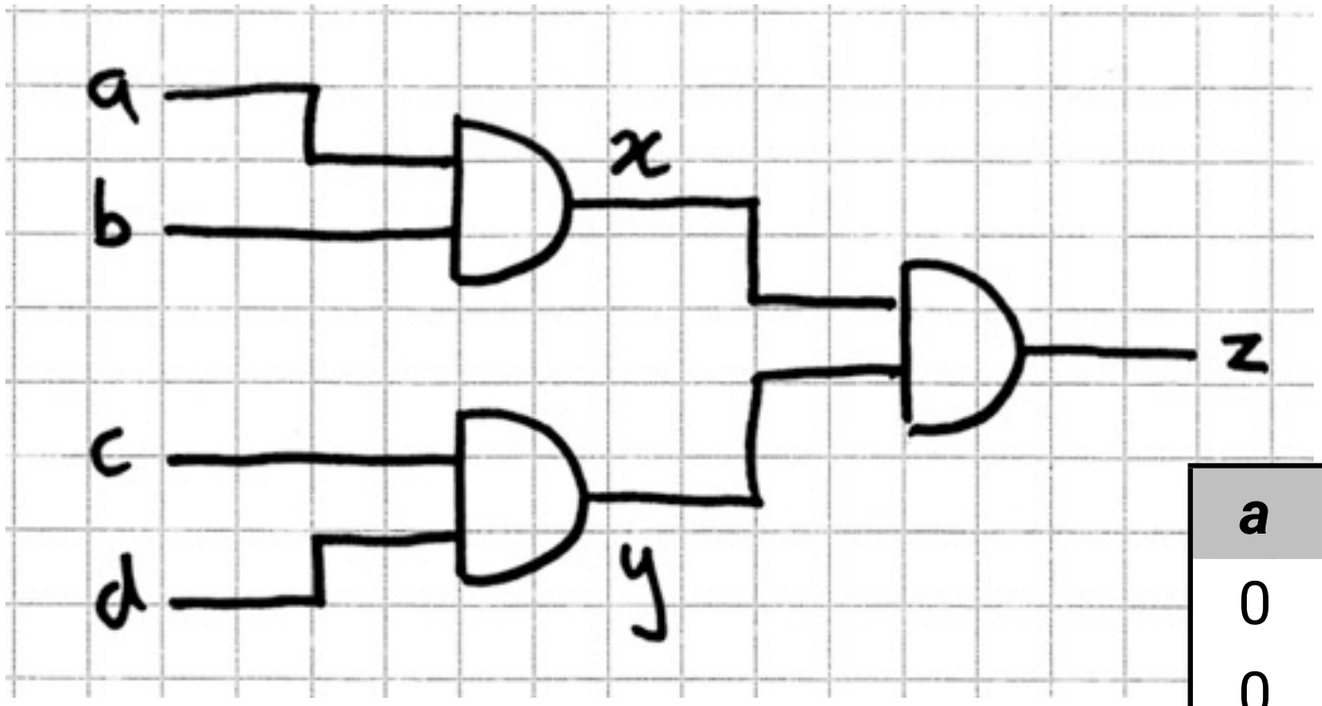


[17.1] Logic gates



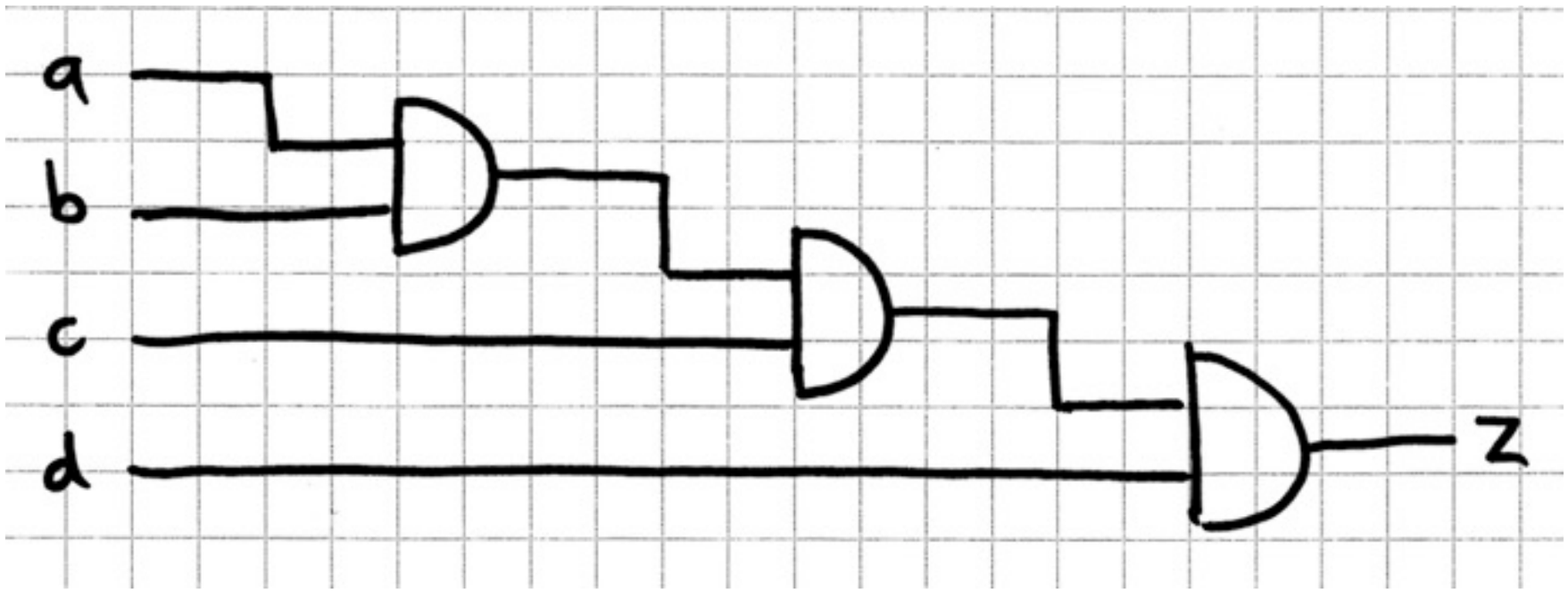
AND		
a	***b***	***z***
0	0	0
0	1	0
1	0	0
1	1	1

Connecting gates



a	*b*	*c*	*d*	*x*	*y*	*z*
0	0	0	0	0	0	0
0	0	0	1	0	0	0
0	0	1	0	0	0	0
0	0	1	1	0	1	0
…	…	…	…	…	…	…
1	1	0	1	1	0	0
1	1	1	0	1	0	0
1	1	1	1	1	1	1

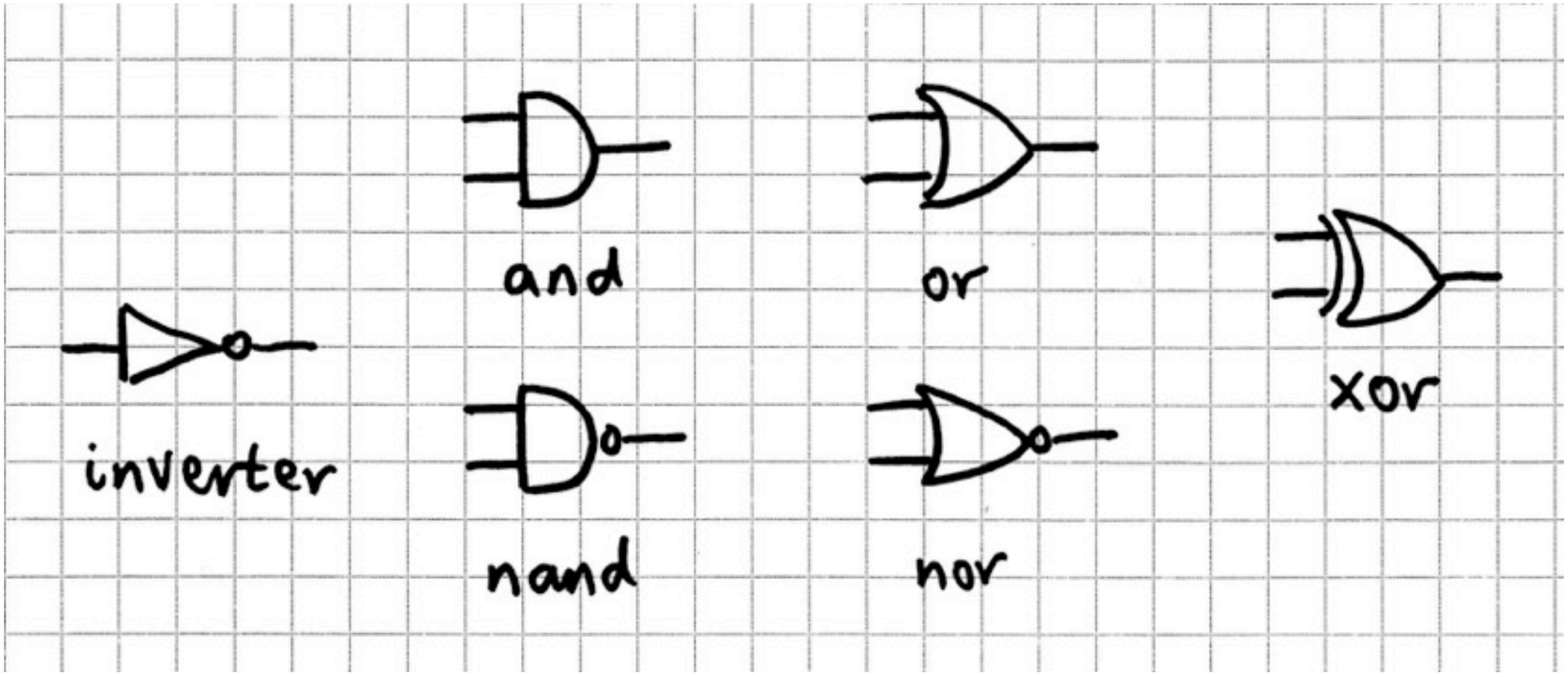
Propagation delay



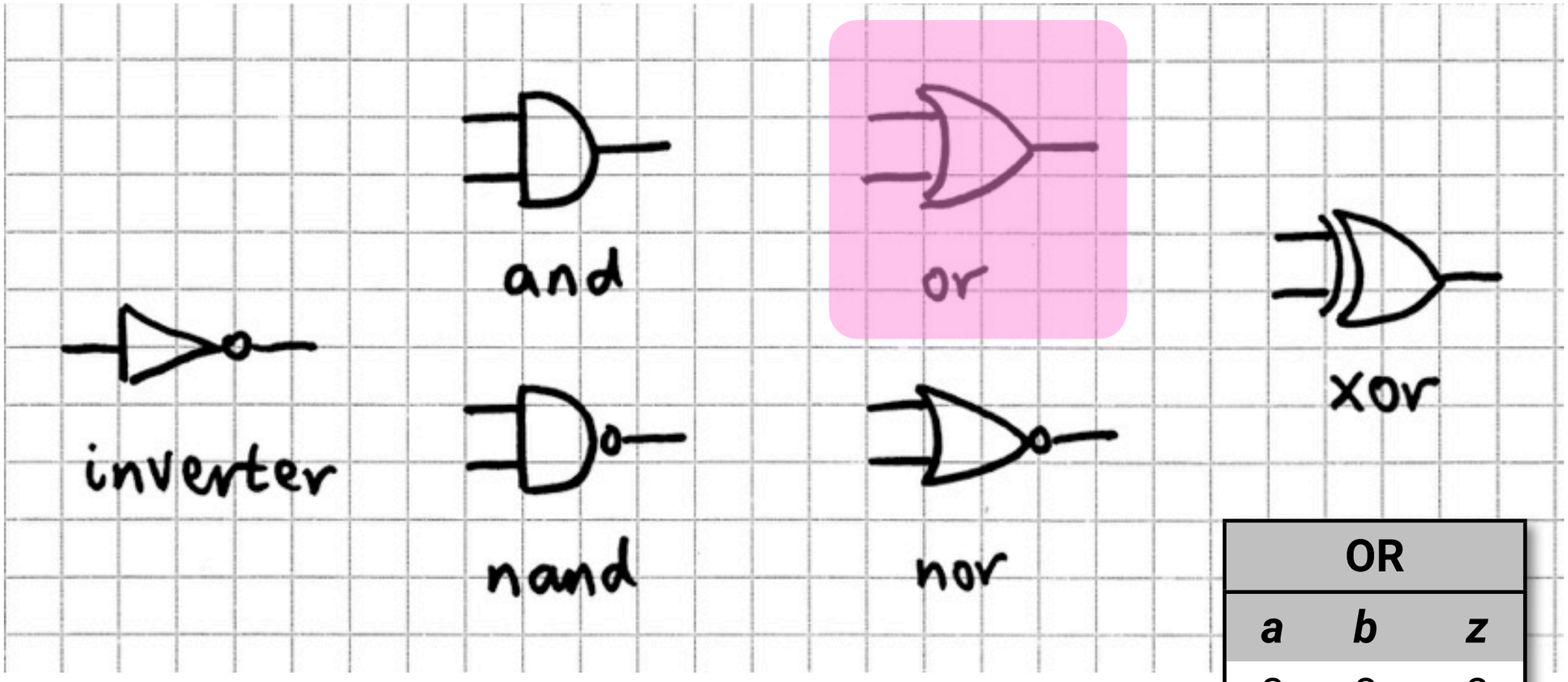
Also a 4-input AND gate

But it performs less well, because delays add up along a longer path.

Other gates

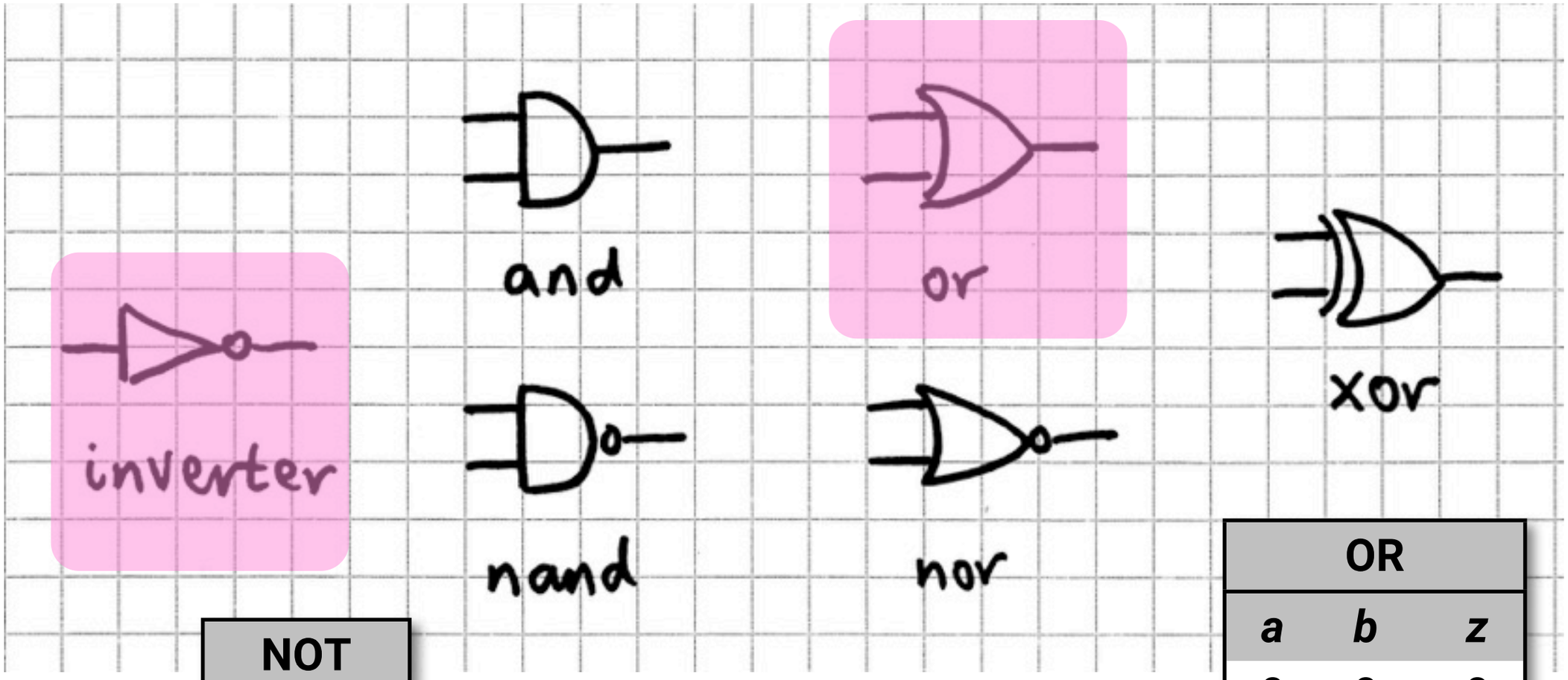


Other gates



OR		
a	***b***	***z***
0	0	0
0	1	1
1	0	1
1	1	1

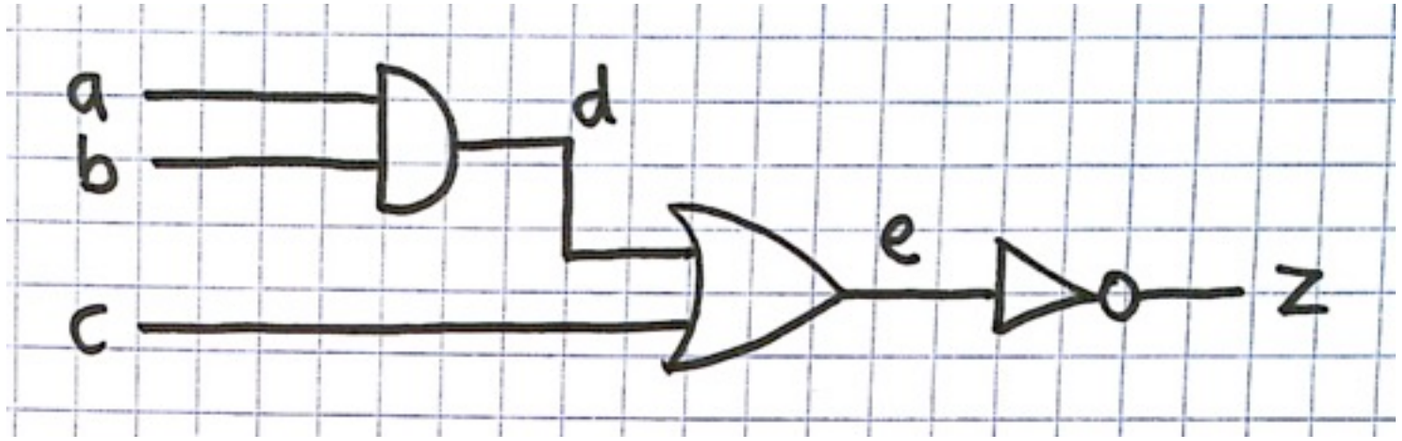
Other gates



NOT	
a	***z***
0	1
1	0

OR		
a	***b***	***z***
0	0	0
0	1	1
1	0	1
1	1	1

Mixed gates



This circuit computes

$$z = \neg((a \wedge b) \vee c).$$

a	*b*	*c*	*d*	*e*	*z*
0	0	0	0	0	1
0	0	1	0	1	0
0	1	0	0	0	1
0	1	1	0	1	0
1	0	0	0	0	1
1	0	1	0	1	0
1	1	0	1	1	0
1	1	1	1	1	0

Adequacy – an example

AND, OR and NOT gates are all we need to compute any Boolean function.

Suppose we want to implement this function.

MAJ			
a	***b***	***c***	***z***
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Adequacy

Write a *product term* for each line where the function is 1.

$p = \neg a \wedge b \wedge c$

$q = a \wedge \neg b \wedge c$

$r = a \wedge b \wedge \neg c$

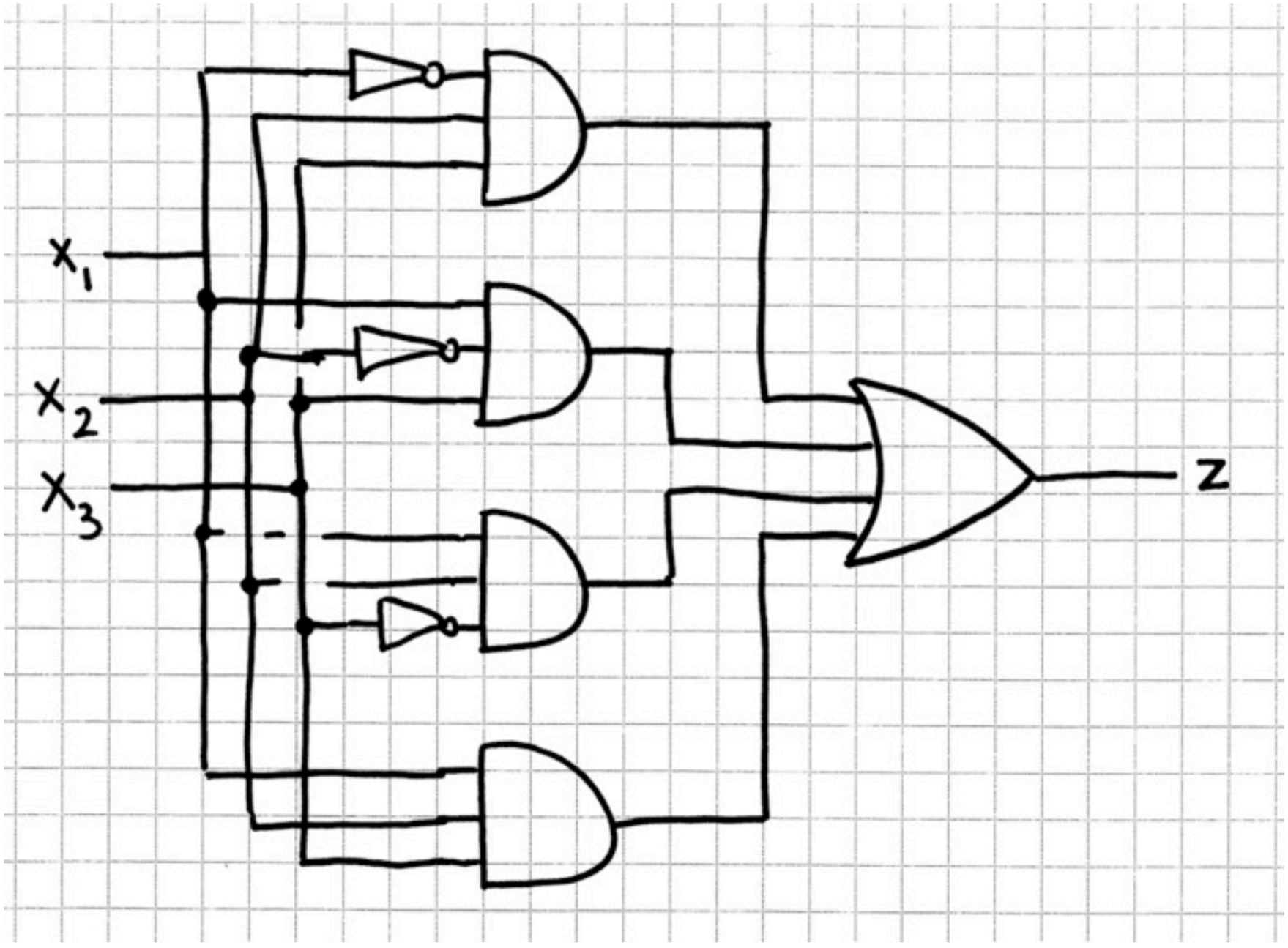
$s = a \wedge b \wedge c$

Take the disjunction of all these terms.

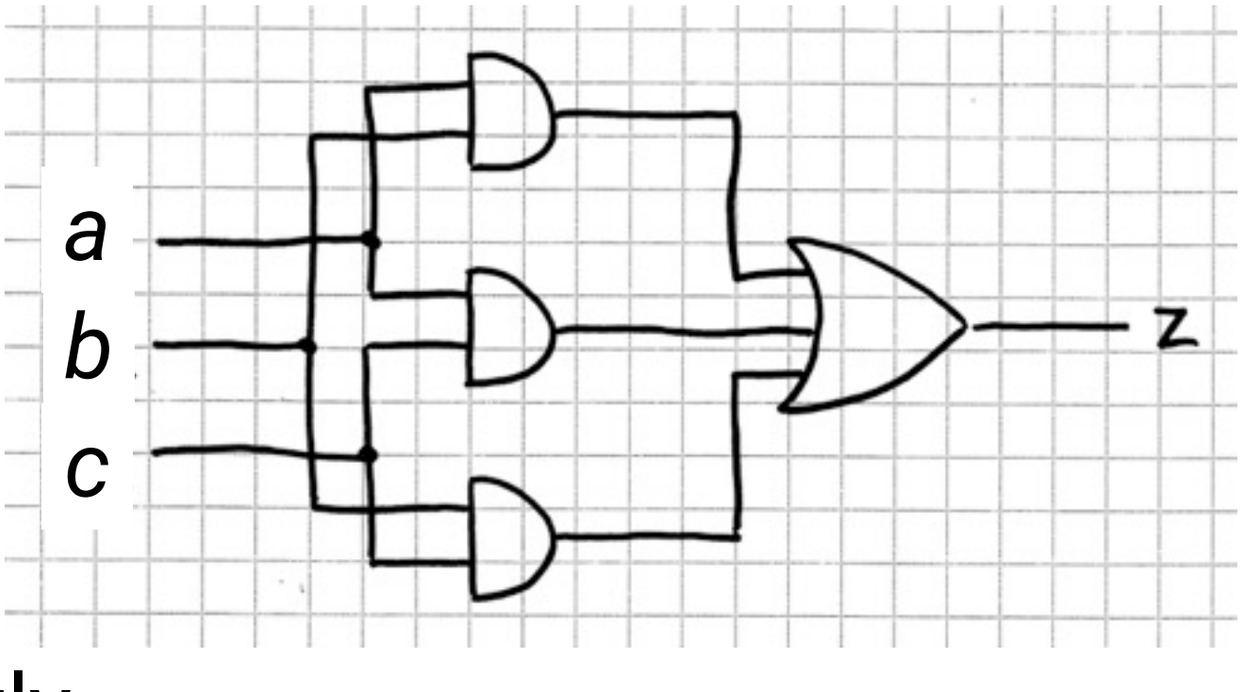
$z = p \vee q \vee r \vee s$

MAJ							
a	***b***	***c***	***p***	***q***	***r***	***s***	***z***
0	0	0					0
0	0	1					0
0	1	0					0
0	1	1	1				1
1	0	0					0
1	0	1		1			1
1	1	0			1		1
1	1	1				1	1

Drawing it as a circuit



A better circuit



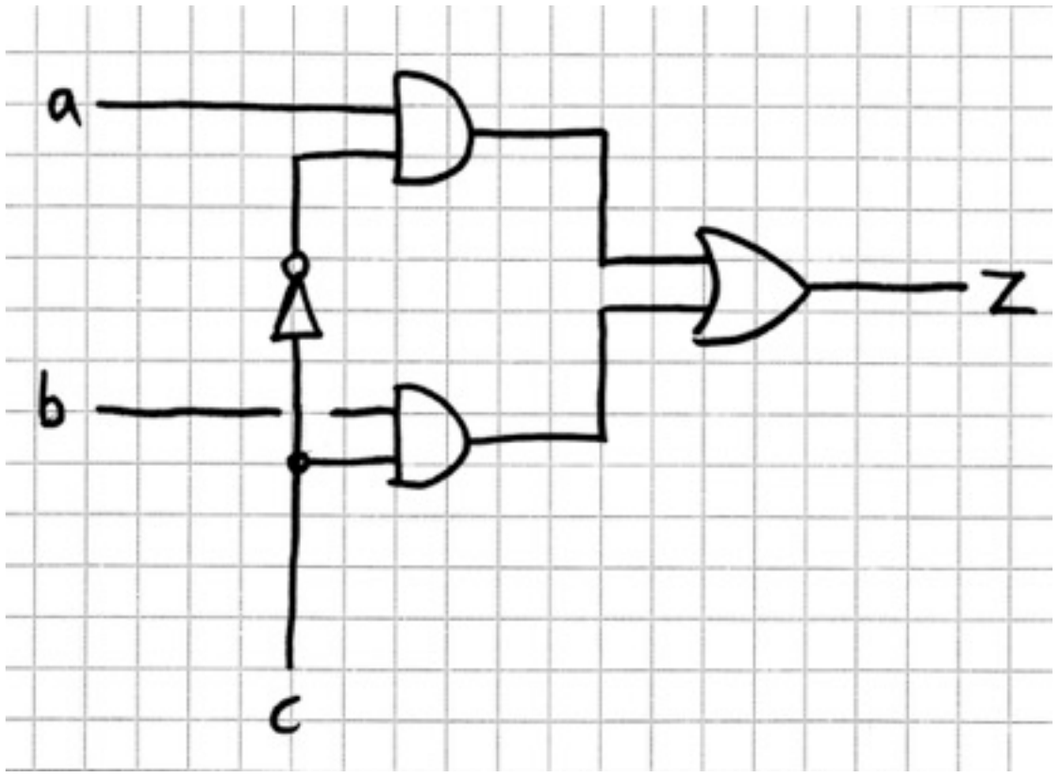
Equivalently,

$$z = (a \wedge b) \vee (b \wedge c) \vee (c \wedge a).$$

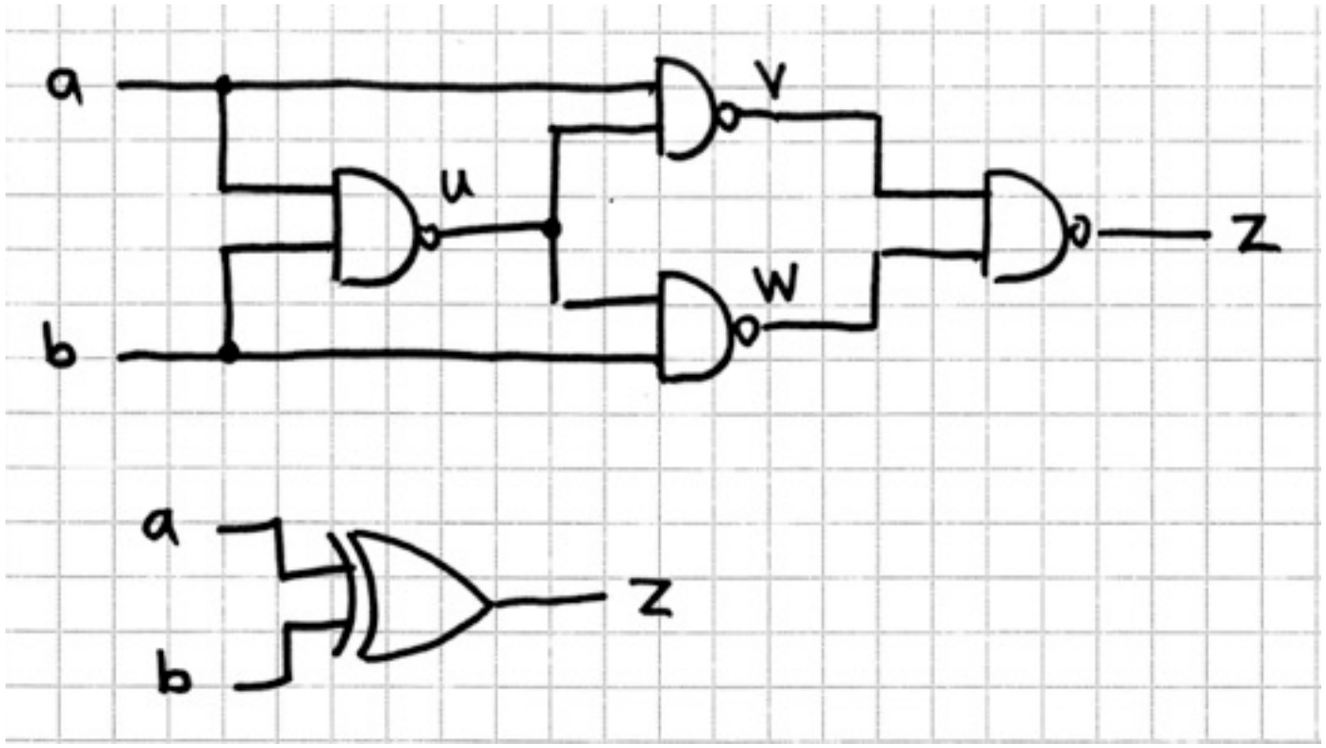
Multiplexer

This *multiplexer* outputs either *a* or *b*, depending on a control input *c*.

$$z = (c\ ?\ b : a)$$



XOR gate



XOR		
a	***b***	***z***
0	0	0
0	1	1
1	0	1
1	1	0

This snazzy implementation has a pleasing symmetry!

Transistors and logic gates

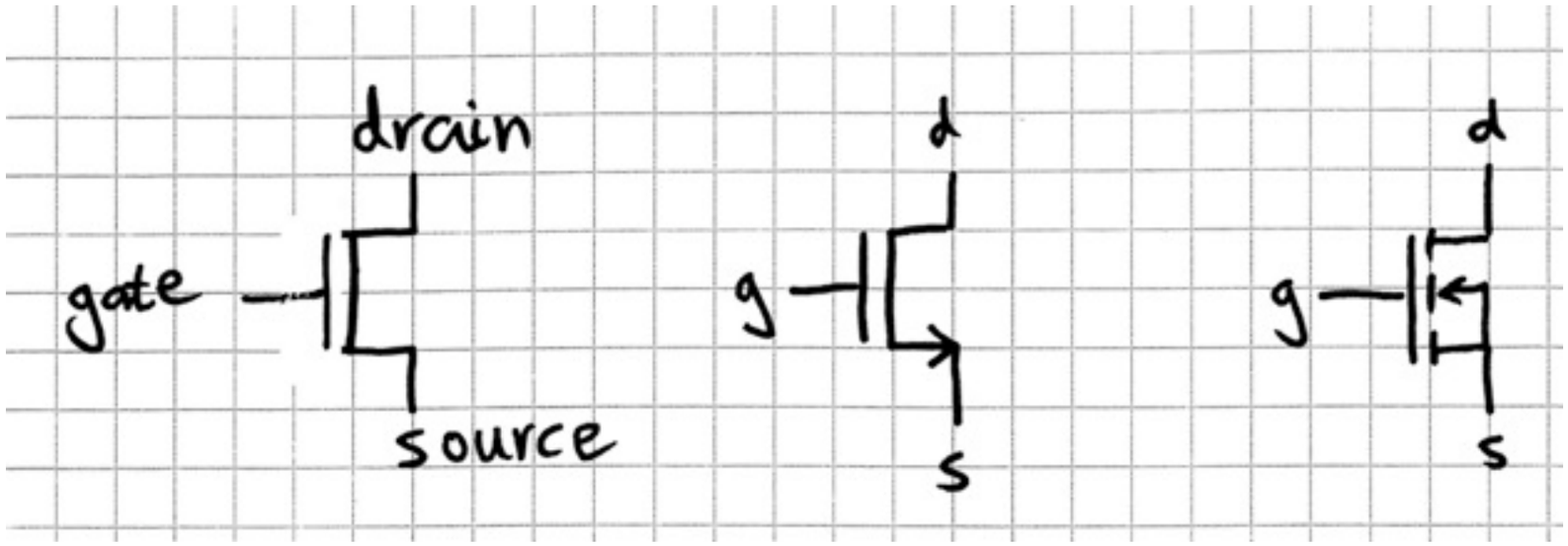
Digital Systems – Lecture 18

Department of
COMPUTER
SCIENCE

In this lecture

- In digital logic, *transistors* behave as switches.
- Logic gates can be designed by combining transistors in fixed patterns.

n-type transistors

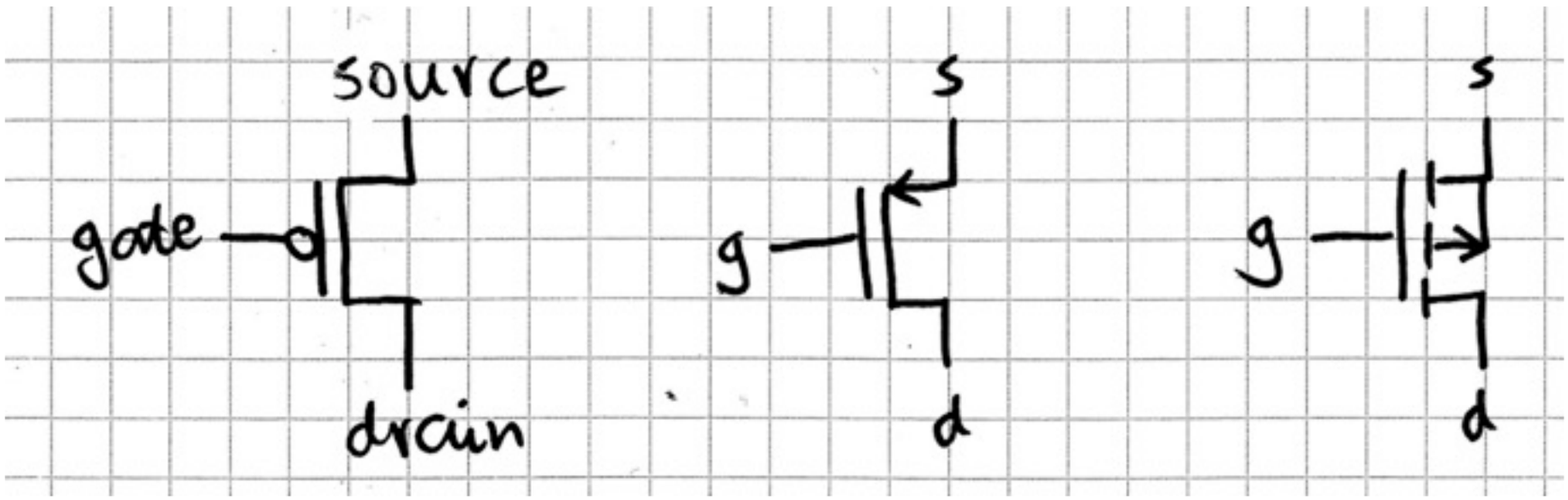


When the gate is at a positive voltage w.r.t. the source, the source and drain are connected.

In a steady state, no current flows at the gate.

We will use the left-hand symbol despite the ambiguity between source and drain.

p-type transistors



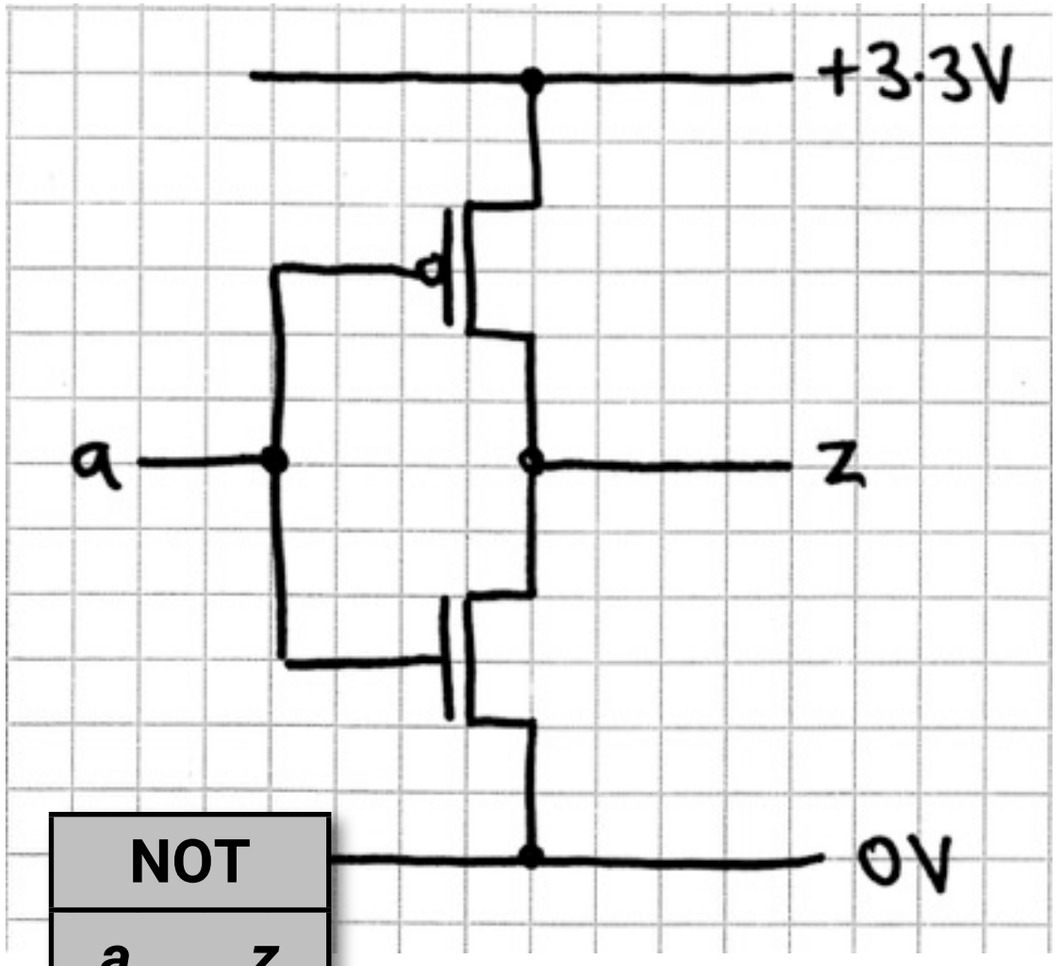
When the gate is at a *negative* voltage w.r.t. the source, the source and drain are connected.

Note that the arrow points towards the channel, and (usually) the source is shown at the top.

CMOS Inverter

When the input *a* is at +3.3V, the *n*-type transistor conducts and pulls the output *z* down to 0V.

When the input *a* is at 0V, the *p*-type transistor conducts and pulls the output *z* up to +3.3V.

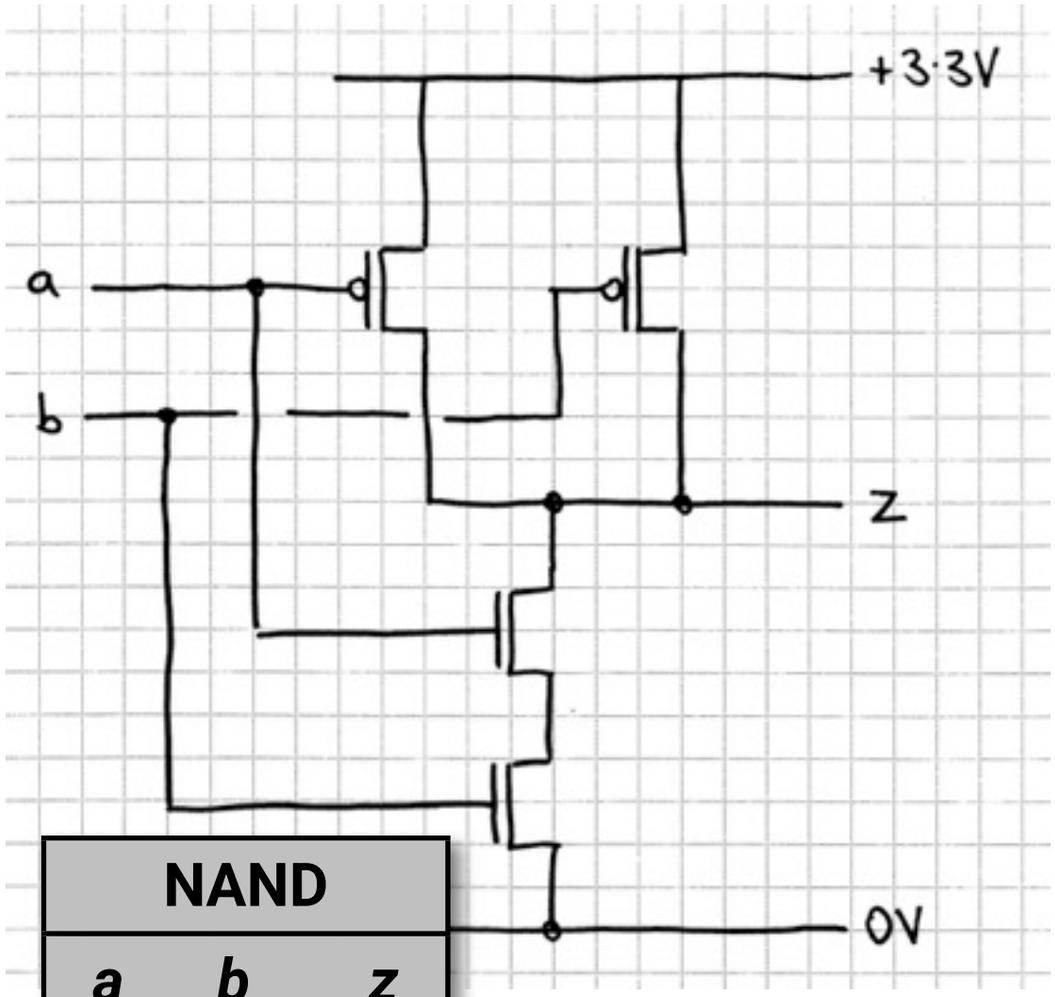


NOT	
a	***z***
0	1
1	0

NAND gate

When both *a* and *b* are high, the two *n*-type transistors conduct and pull the output to ground.

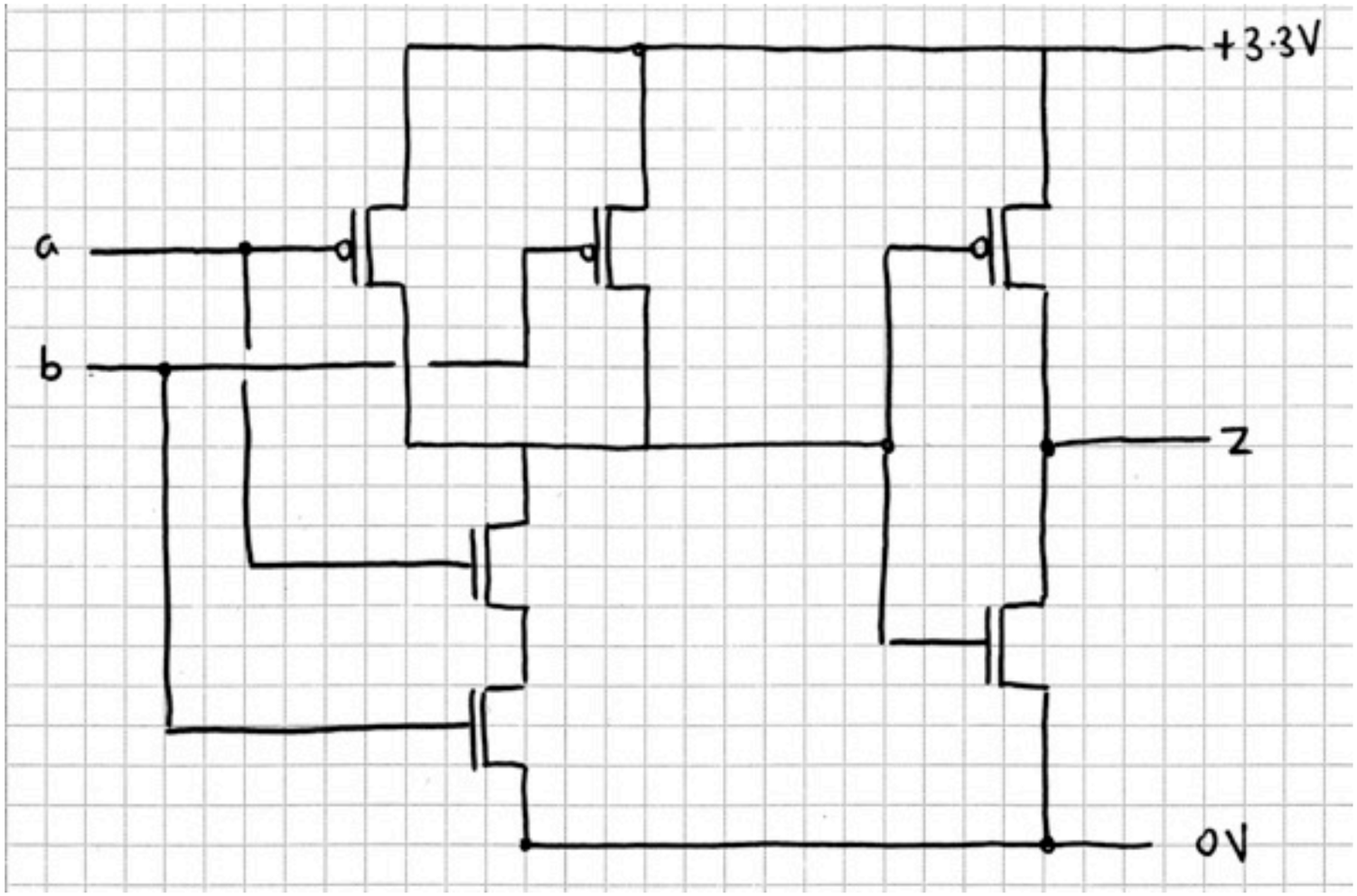
When *a* or *b* or both are low, one or both of the *p*-type transistors conduct and pull the output high.



NAND		
a	***b***	***z***
0	0	1
0	1	1
1	0	1
1	1	0

AND gate

For an AND gate, we must follow the NAND gate with an inverter.

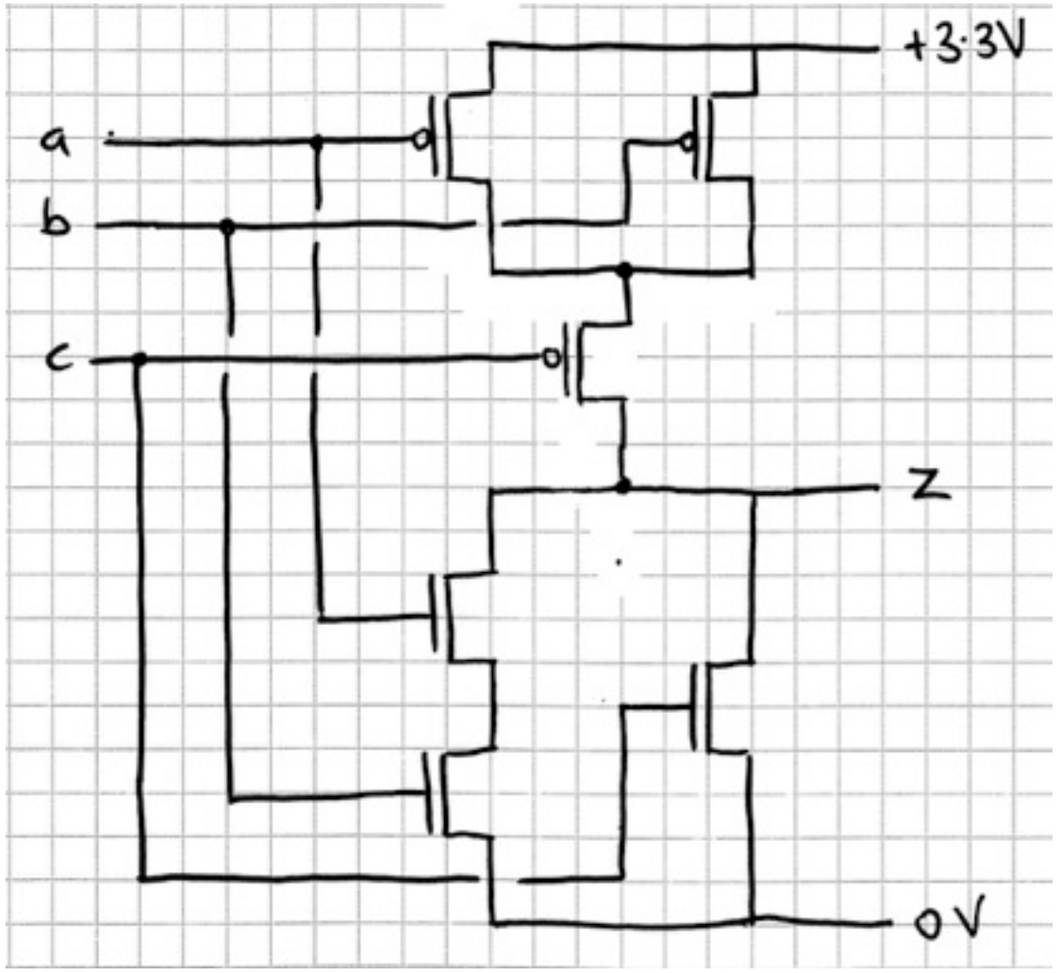


CMOS gates in general

This gate computes

$z = \neg((a \wedge b) \vee c)$

AND-OR-NOT			
a	***b***	***c***	***z***
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

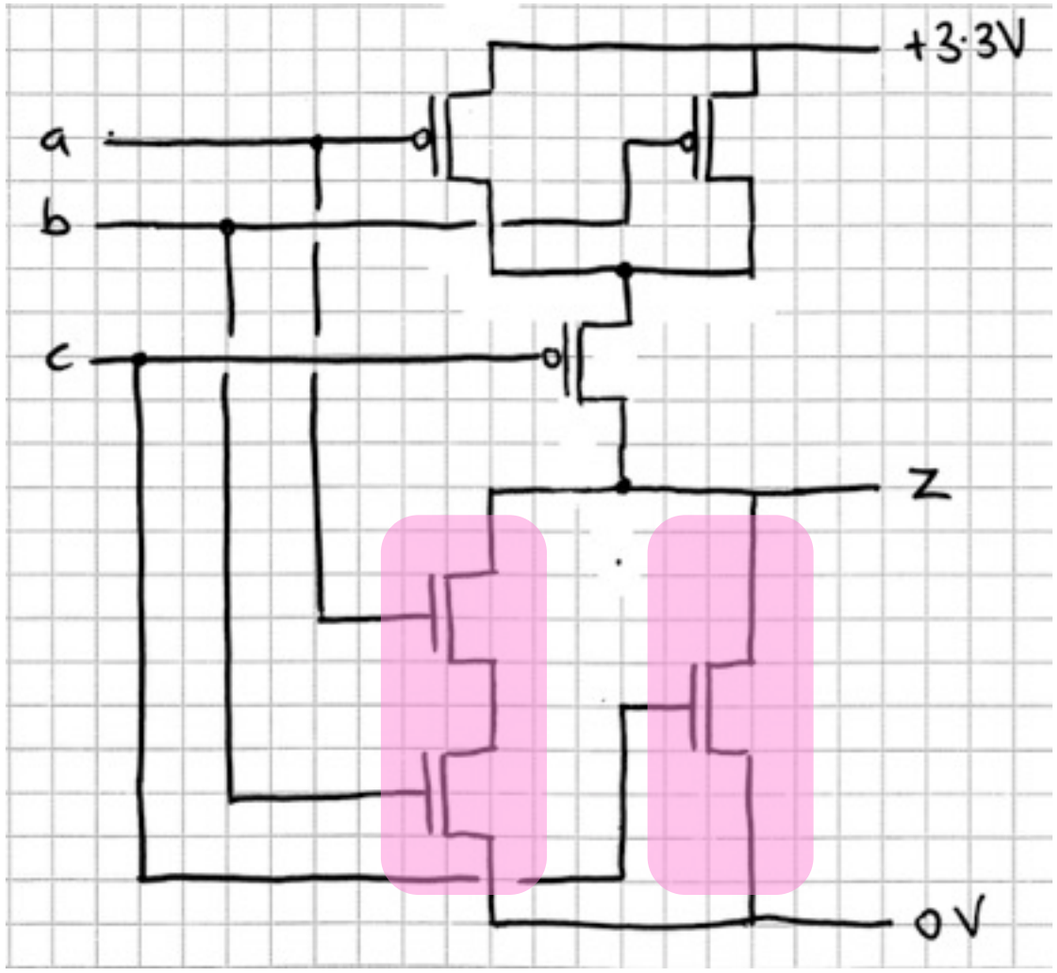


Series and parallel

The pull-down network has elements for {*a*, *b*} and for {*c*} in parallel.

So the pull-up network puts complementary elements in series.

Transistors {*a*, *b*} are in series below, so they are in parallel above.

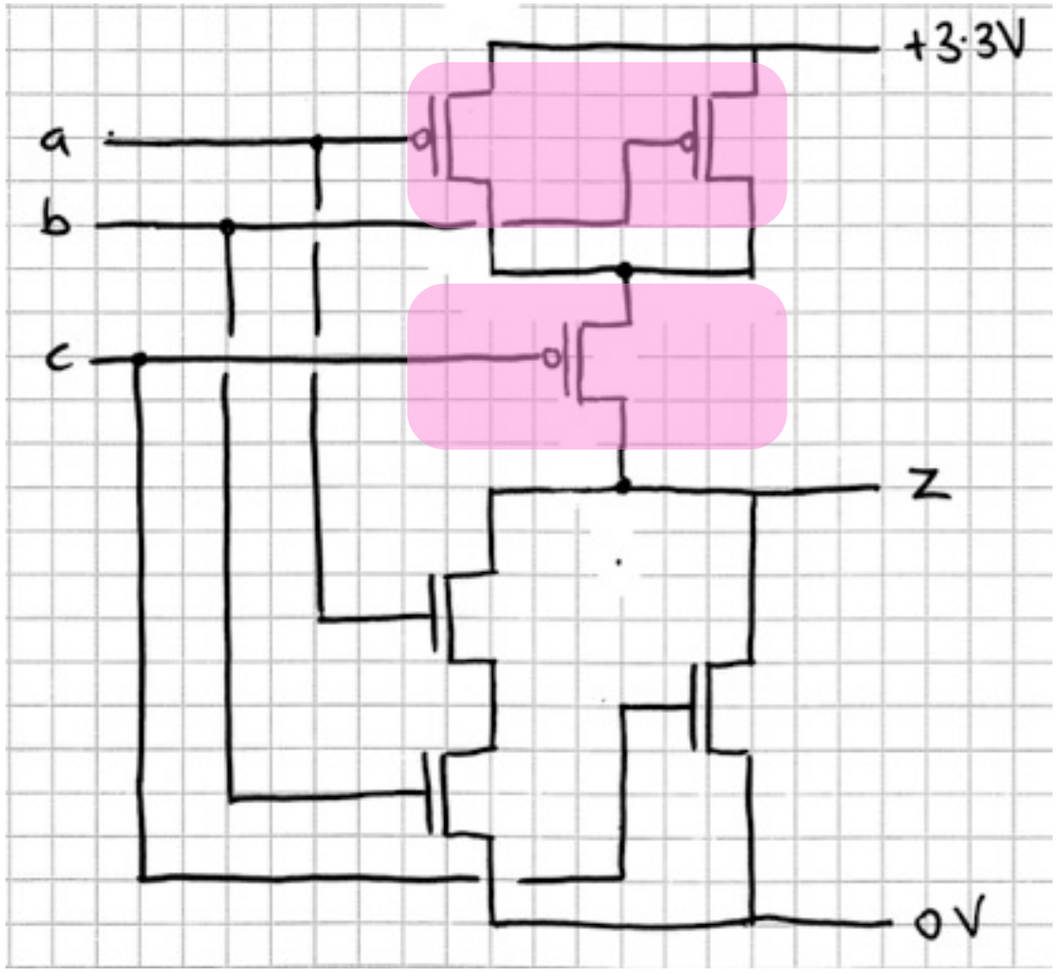


Series and parallel

The pull-down network has elements for {*a*, *b*} and for {*c*} in parallel.

So the pull-up network puts complementary elements in series.

Transistors {*a*, *b*} are in series below, so they are in parallel above.

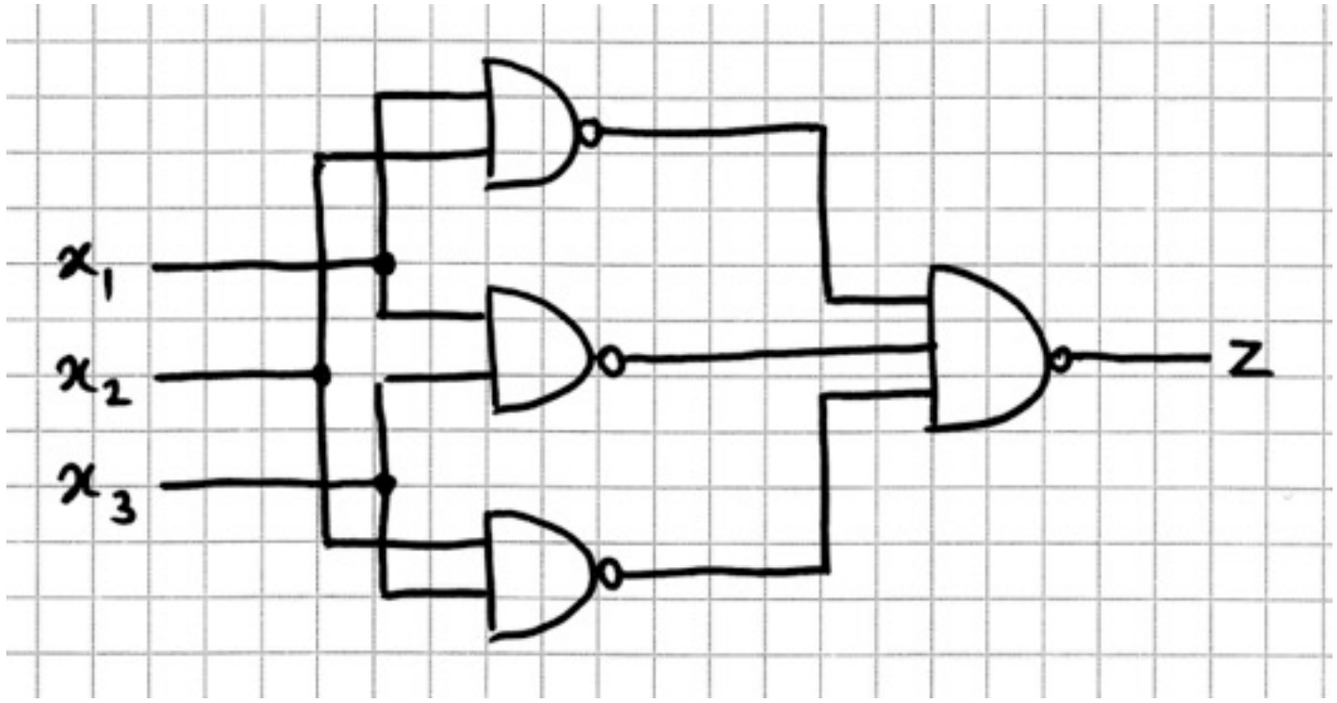


But just NAND is enough

For example,

$z = (a \wedge b) \vee (b \wedge c) \vee (c \wedge a)$

$= \text{NAND}(\text{NAND}(a, b), \text{NAND}(b, c), \text{NAND}(c, a))$



Sequential logic

Digital Systems – Lecture 19

Department of
COMPUTER
SCIENCE

In this lecture

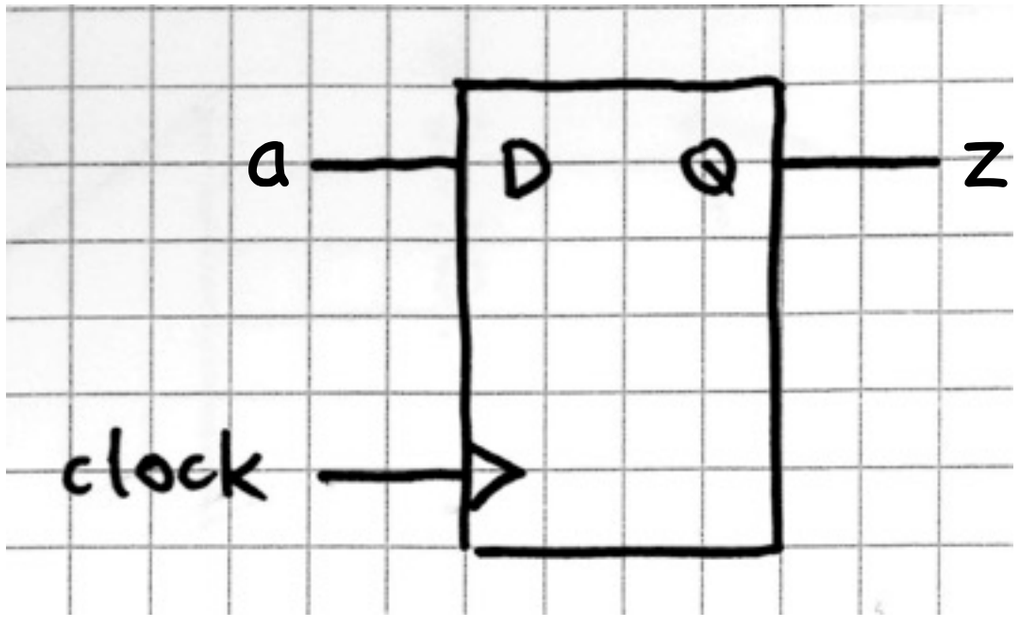
- *Sequential circuits* have outputs that depend on the history of their inputs.
- A *flip-flop* is a primitive element that outputs the previous value of its input.
- Any sequential circuit can be built by combining flip-flops with combinational logic.

D-type flip-flop

In a fully synchonous design, a D-type satisfies

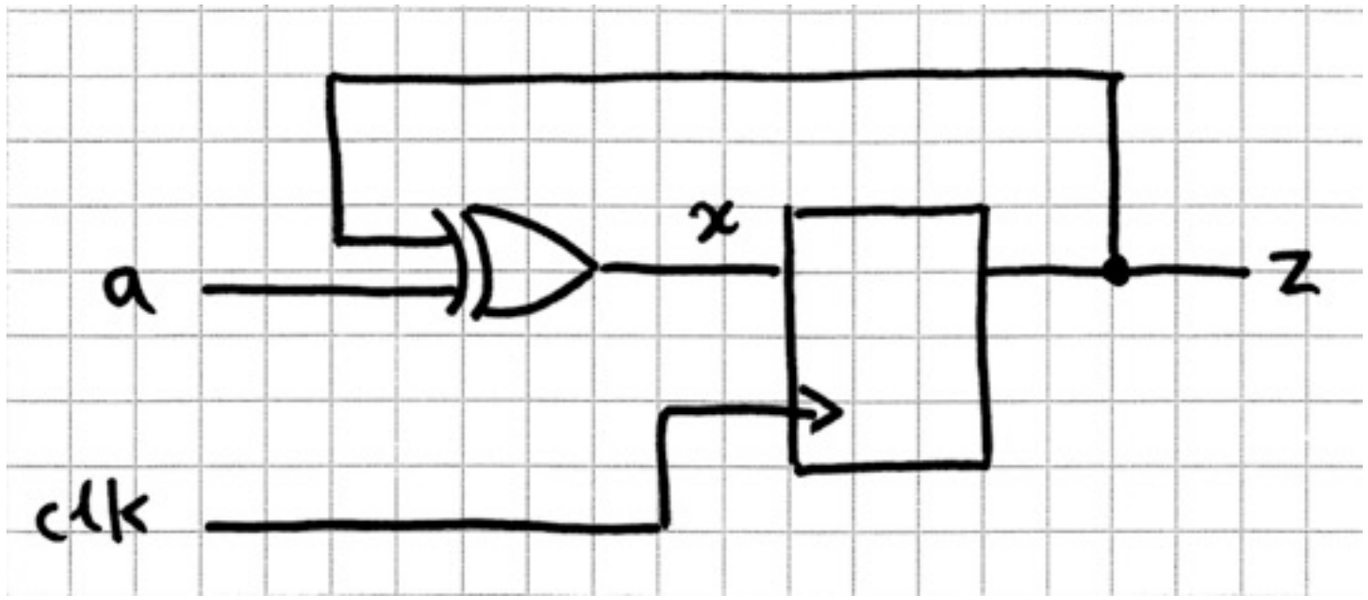
$z_{t+1} = a_t$

(By *fully synchronous* we mean all clocks connected to a common source.)



D-type		
a_t	z_t	z_{t+1}
0	0	0
0	1	0
1	0	1
1	1	1

Parity detector



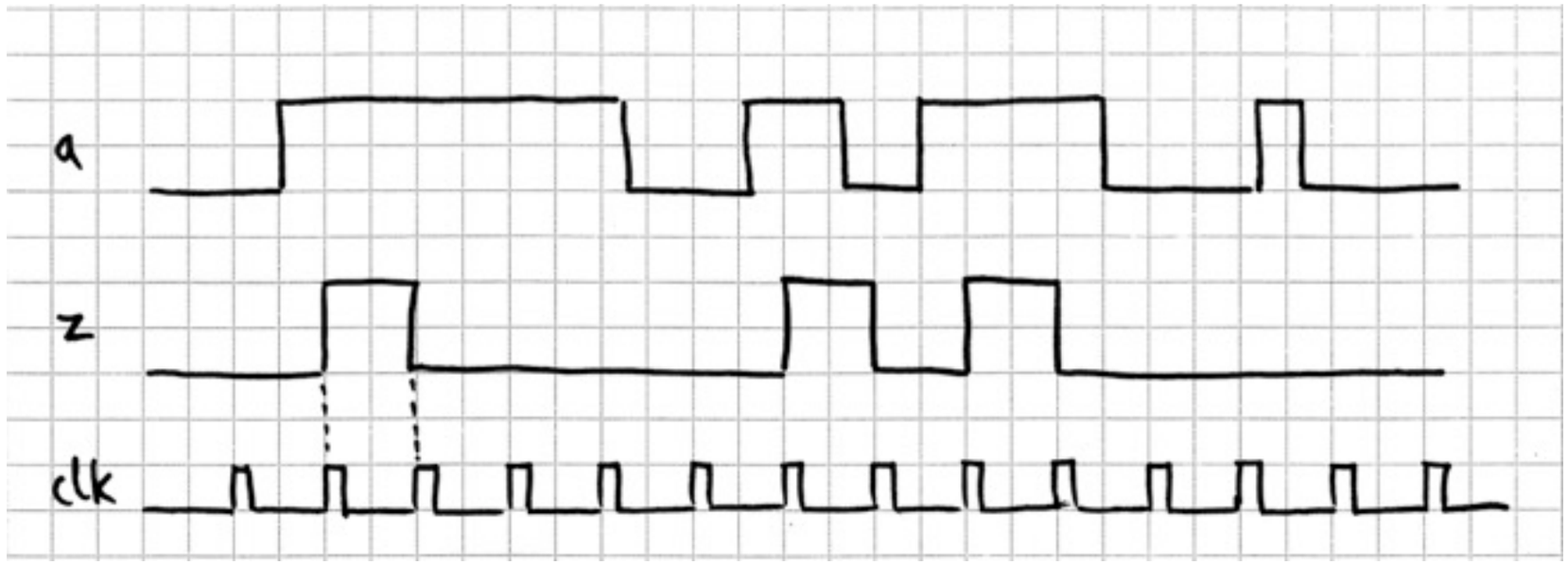
This circuit satisfies

$x_t = a_t \oplus z_t, \quad z_{t+1} = x_t, \quad z_0 = 0$

So (by induction)

$z_t = a_0 \oplus a_1 \oplus \ldots \oplus a_{t-1}.$

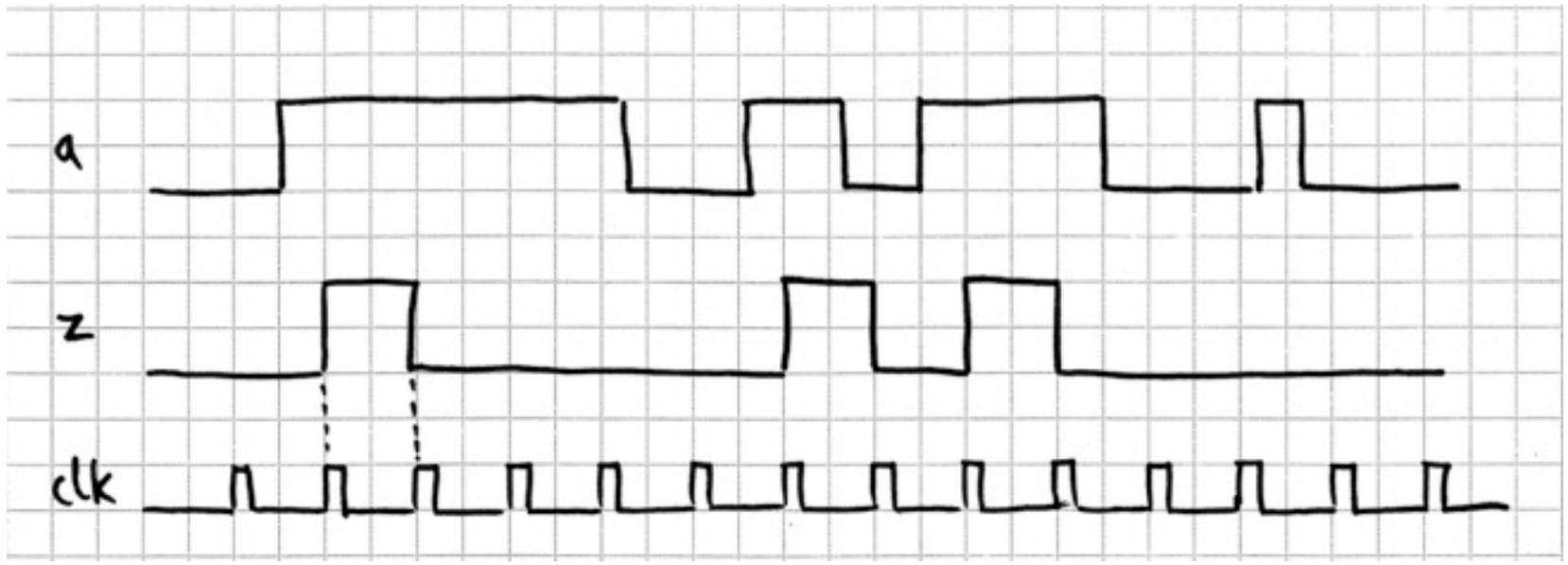
A pulse shaper



Wanted: a circuit that shapes the input into neat pulses, each one clock-cycle long.

Note: we can’t avoid missing pulses that don’t span a clock edge.

Observation



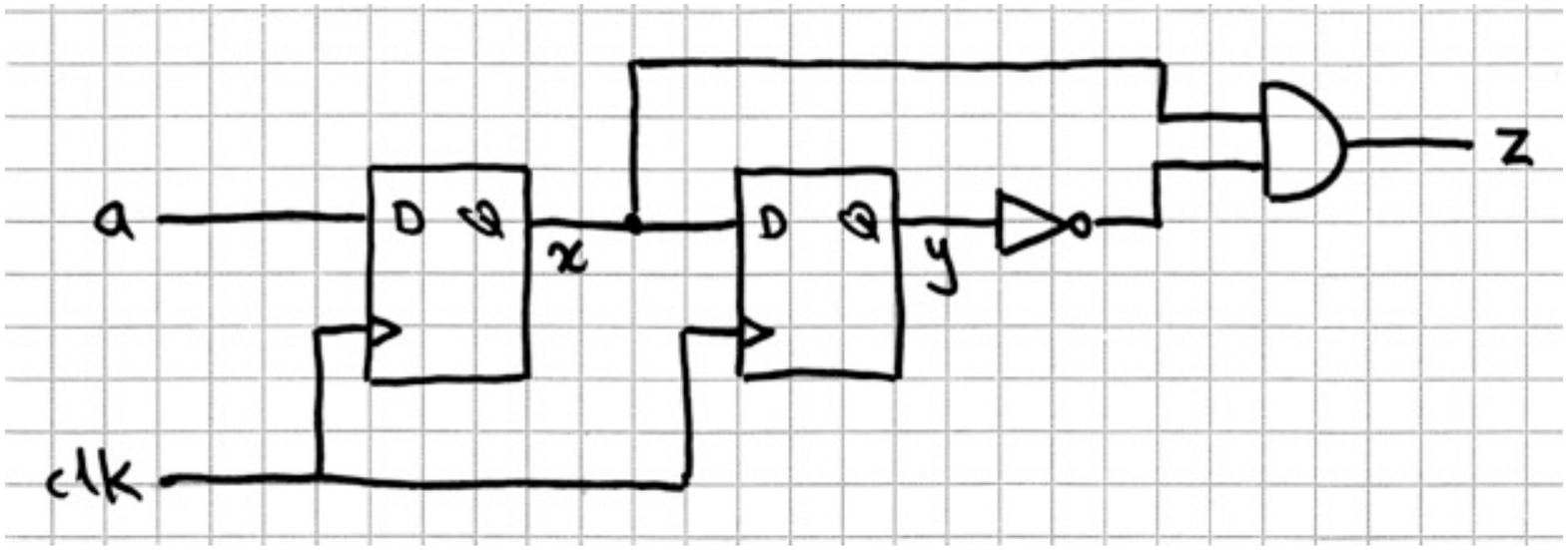
The output is high exactly when the input was high at the last clock-edge but low at the one before it.

If $x_t = a_{t-1}$ and $y_t = a_{t-2}$, then $z_t = x_t \wedge \neg y_t$.

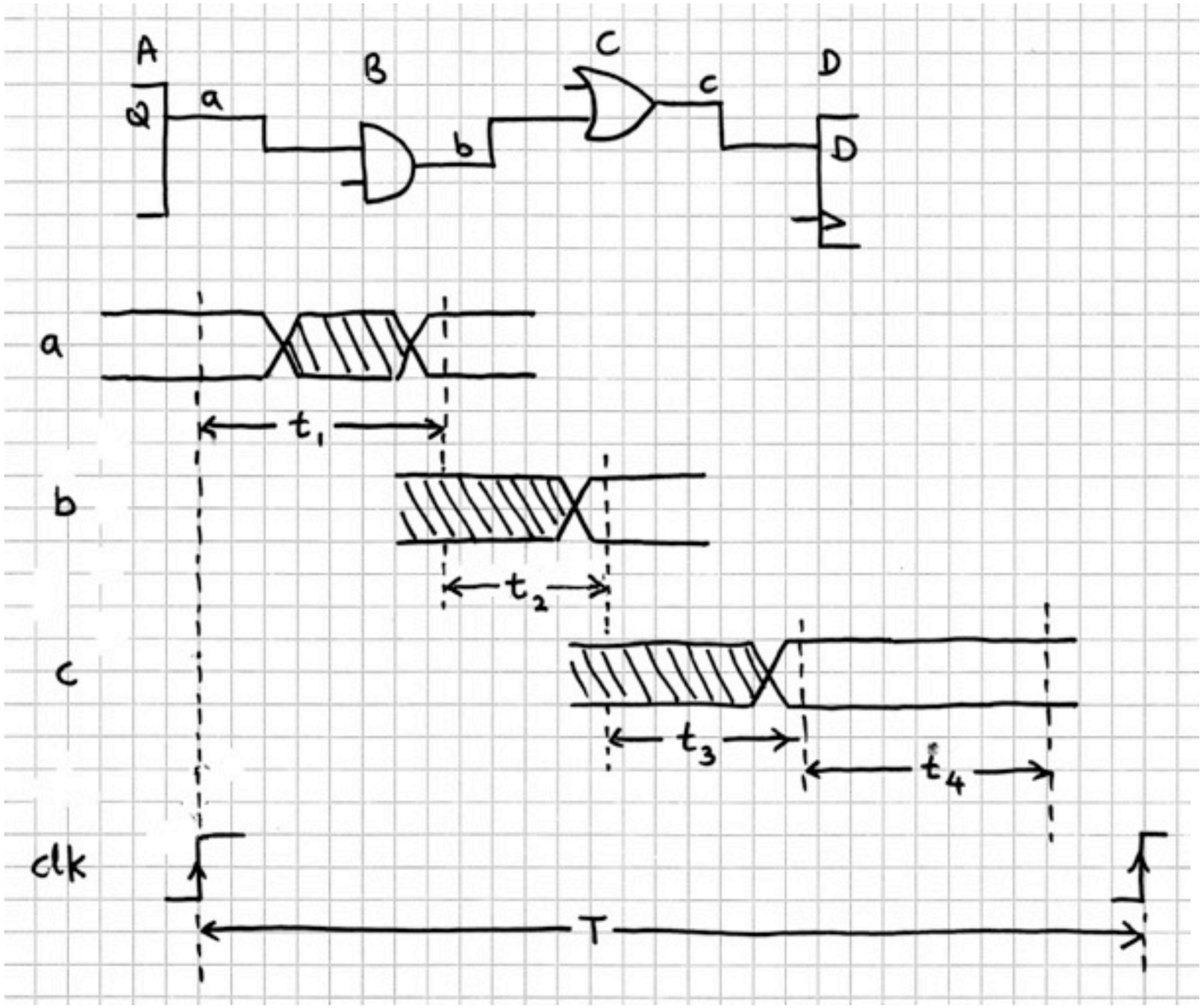
Making a circuit

The output is high exactly when the input was high at the last clock-edge but low at the one before it.

If $x_t = a_{t-1}$ and $y_t = a_{t-2}$, then $z_t = x_t \wedge \neg y_t$.



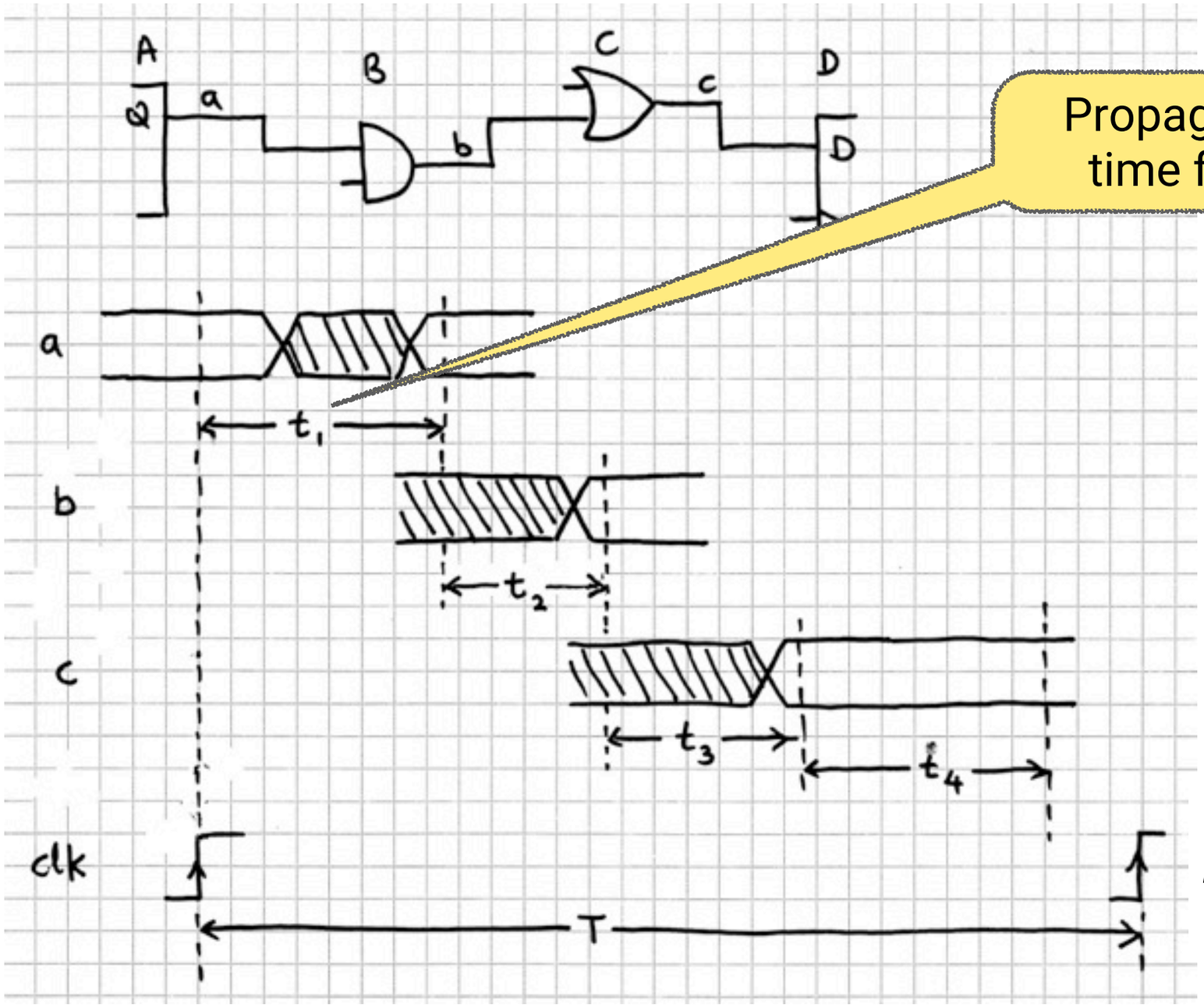
What limits the clock speed?



Must have

$T \geq t_1+t_2+t_3+t_4$

What limits the clock speed?

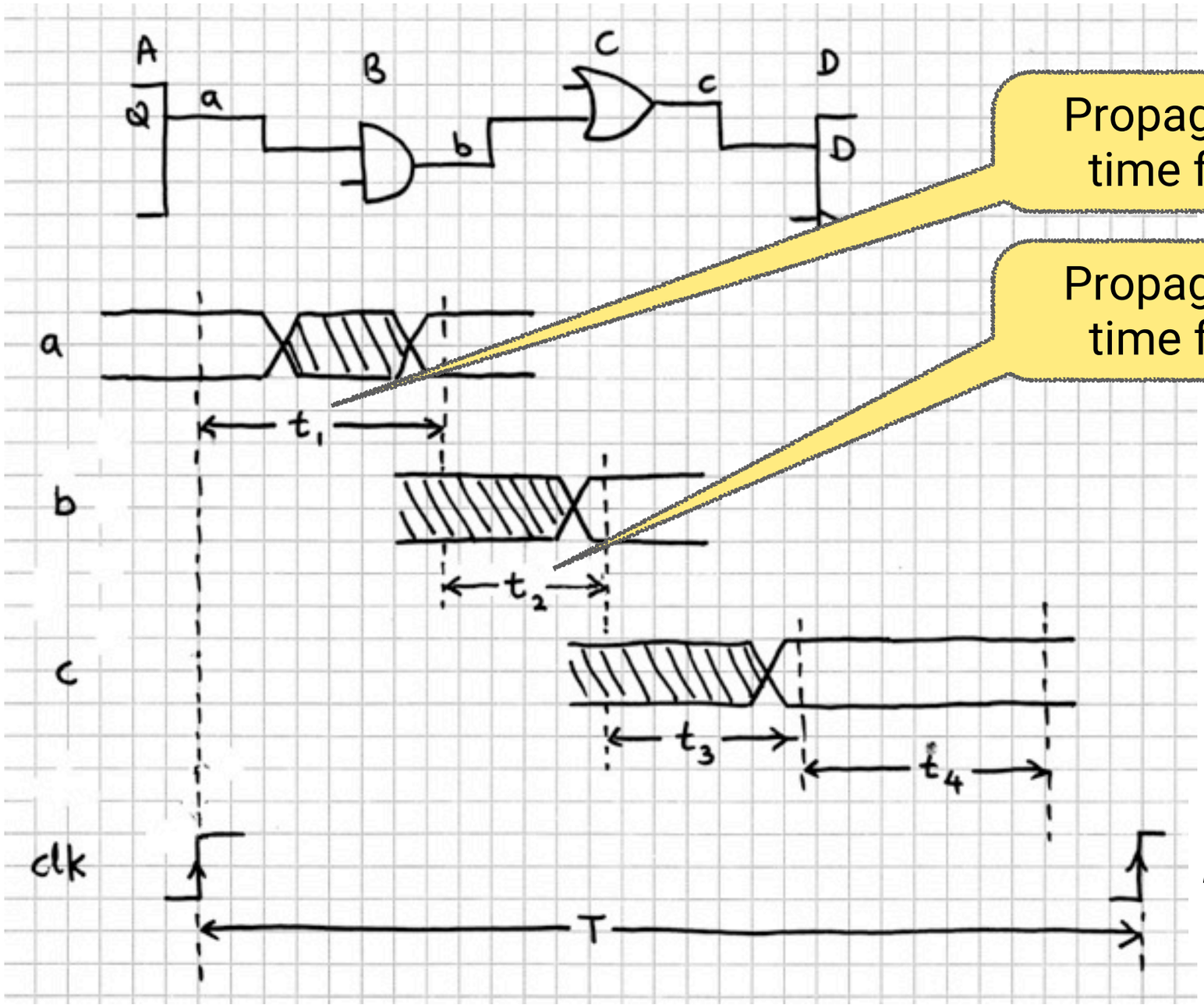


Propagation time for A

Must have

$T \geq t_1+t_2+t_3+t_4$

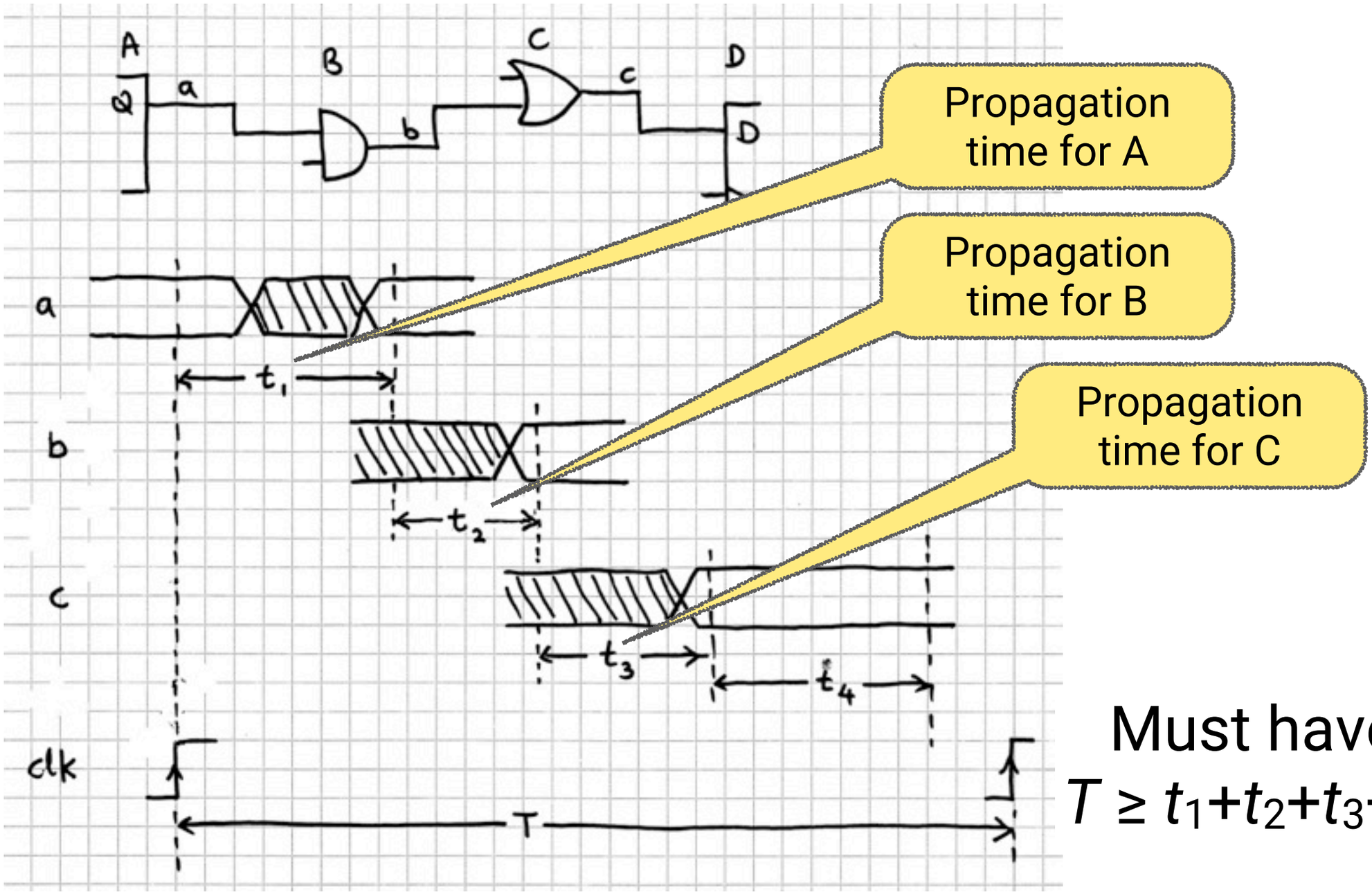
What limits the clock speed?



Must have

$T \geq t_1+t_2+t_3+t_4$

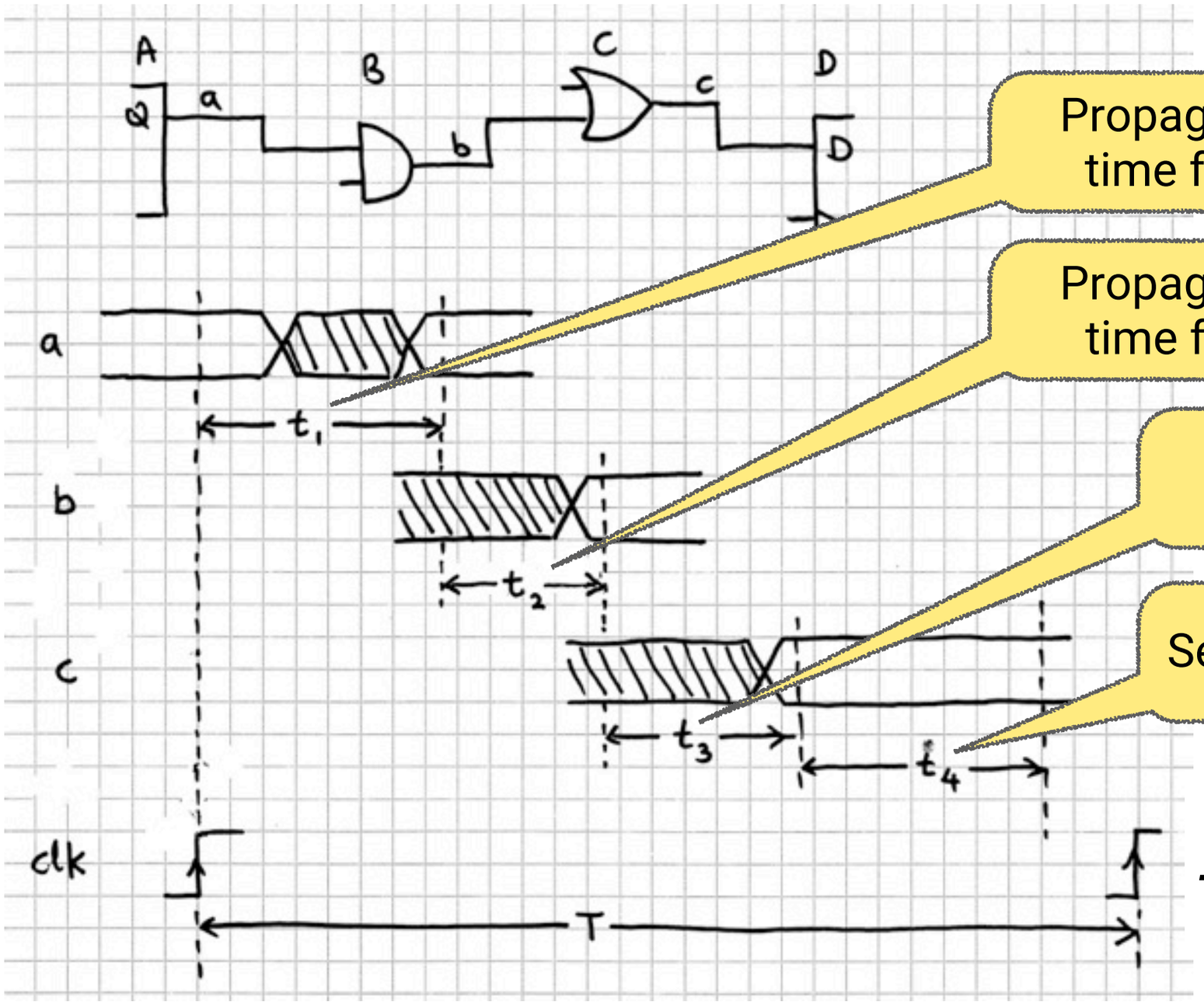
What limits the clock speed?



Must have

$T \geq t_1+t_2+t_3+t_4$

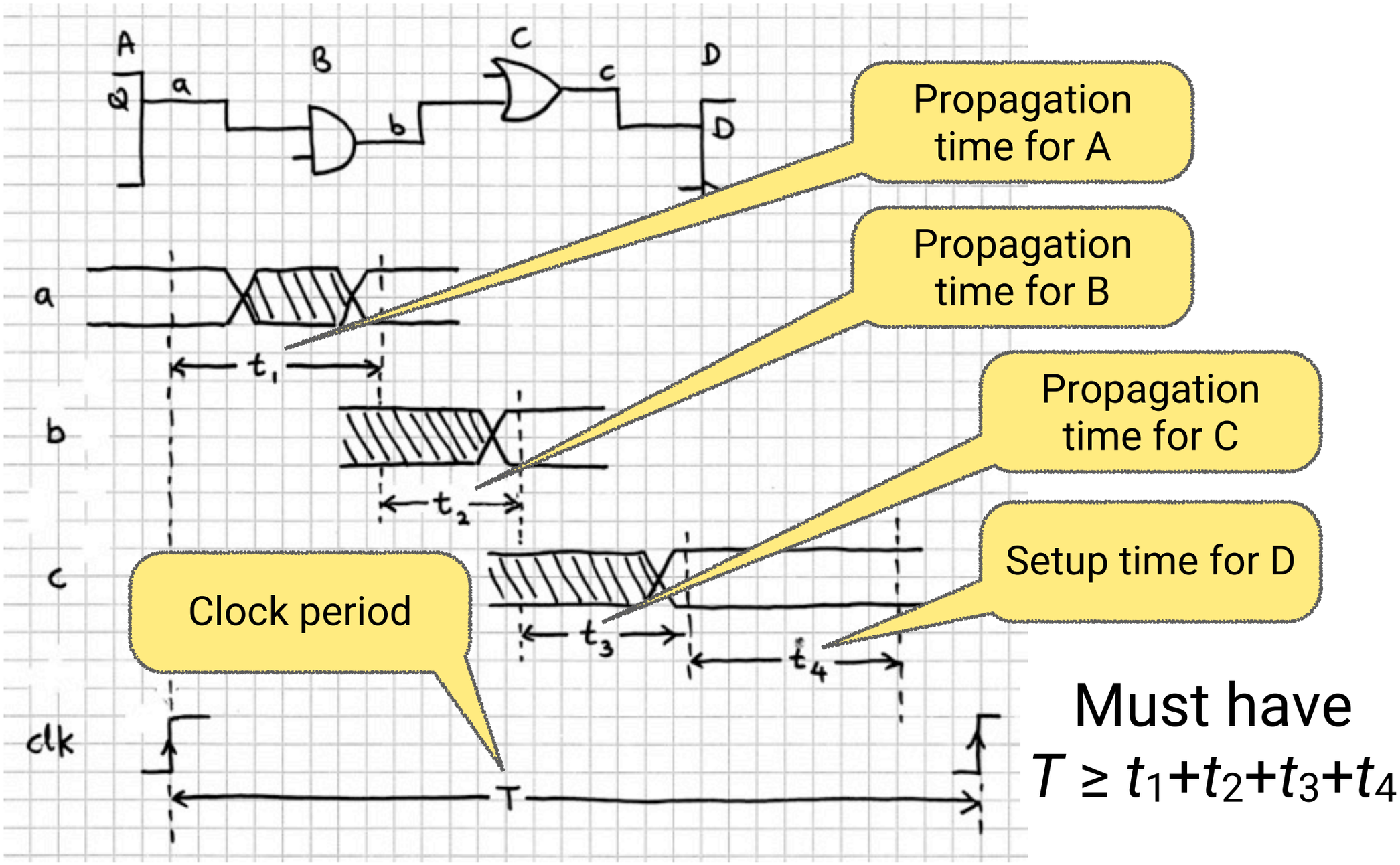
What limits the clock speed?



Must have

$T \geq t_1+t_2+t_3+t_4$

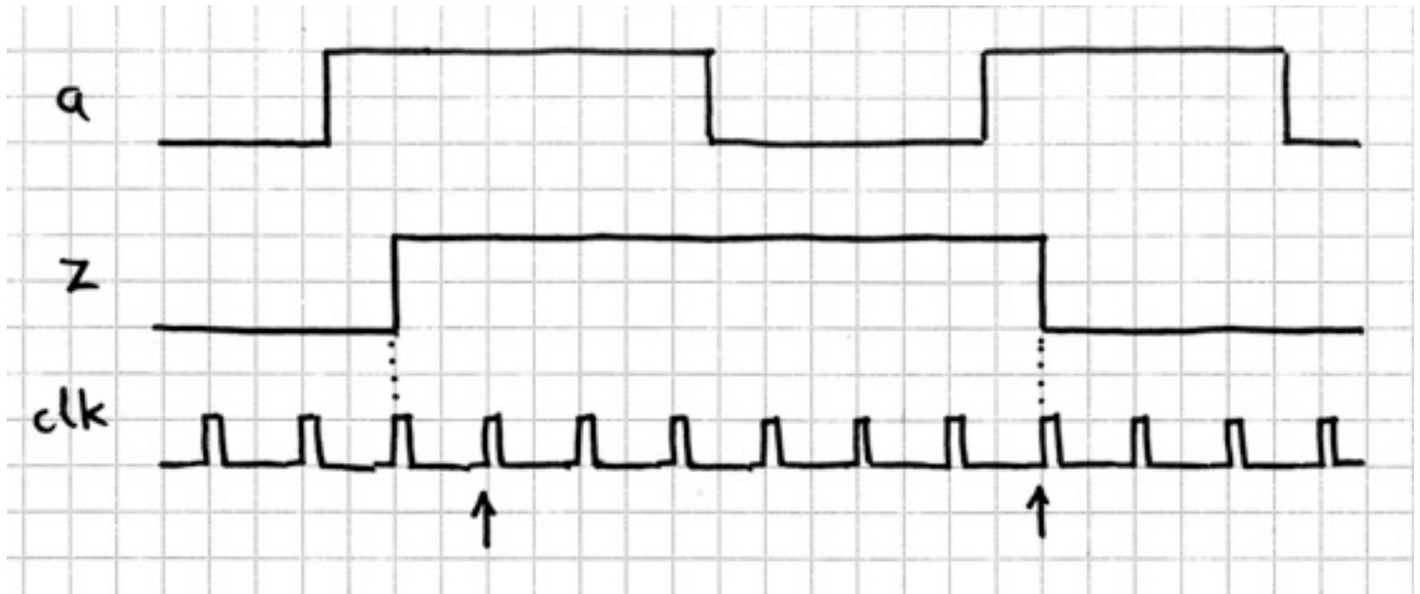
What limits the clock speed?



Must have

$T \geq t_1+t_2+t_3+t_4$

A bathroom light switch



The light *z* changes state once for each pull of the cord *a*.

Notice there are *four* states, because *a* = 1 and z = 1 at both places marked, and yet the next state is different.

Writing a ‘program’

When the cord is pulled, *x* remembers the state before the pull, and *y* is set opposite to it.

When the cord is released, *y* is continually copied to *x* ready for the next pull.

The output is *y*.

```
x = y = 0;
forever {
    z = y;
    wait for clock;
    if (a)
        y = ¬x;
    else
        x = y;
}
```

Simultaneous update

If *a* is true, then

$(x_{t+1}, y_{t+1}) = (x_t, \neg x_t)$.

If *a* is false, then

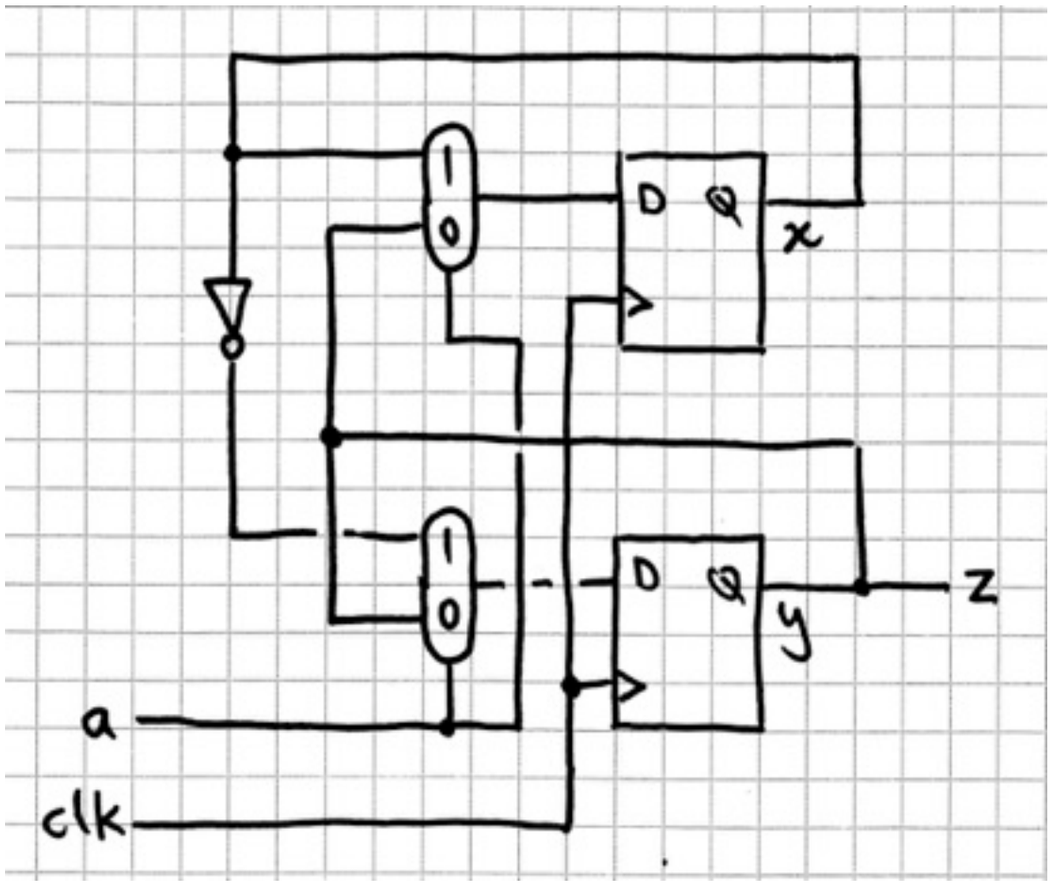
$(x_{t+1}, y_{t+1}) = (y_t, y_t)$.

So

$x_{t+1} = (a\ ?\ x_t : y_t)$
$y_{t+1} = (\mathrm{a}\ ?\ \neg x_t : y_t)$

```
x = y = 0;
forever {
    z = y;
    wait for clock;
    if (a)
        y = ¬x;
    else
        x = y;
}
```

Making a circuit



The two ovals are *multiplexers*, implementing the function MUX(*a*, *b*, *c*) = (*a* ? *b* : *c*).

Architectural elements

Digital Systems – Lecture 20

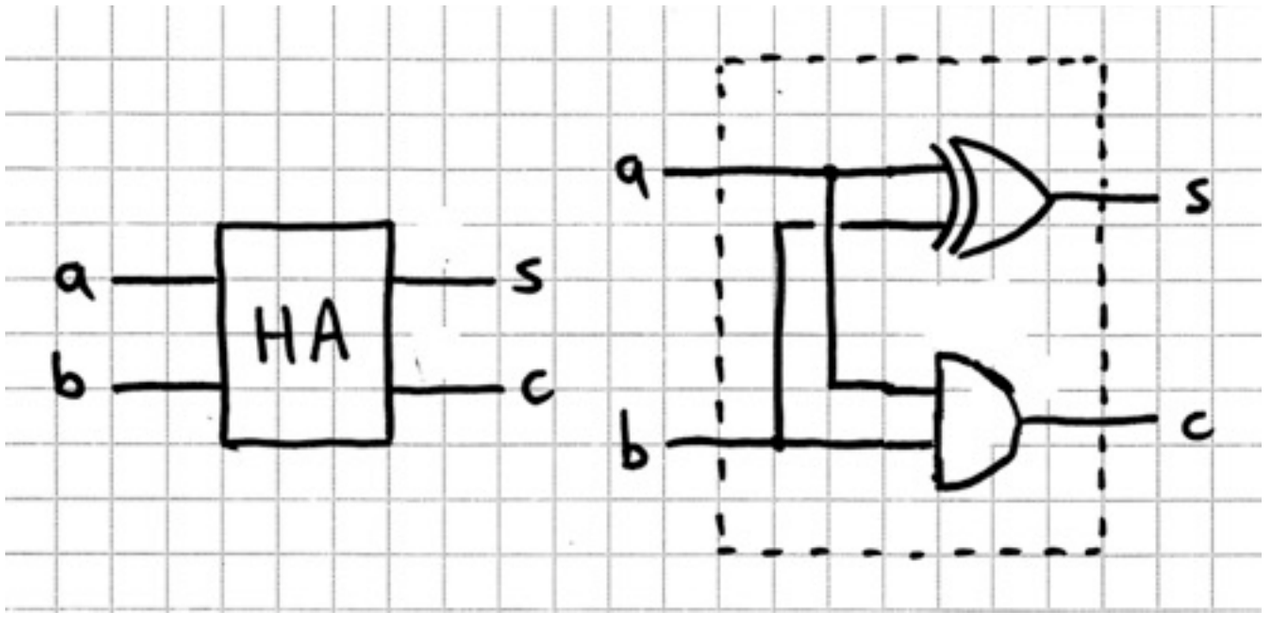
Department of
COMPUTER
SCIENCE

In this lecture

A catalog of digital modules useful in implementing a processor:

- Adders
- Multiplexers
- Registers and register files
- Shifters

Half adder

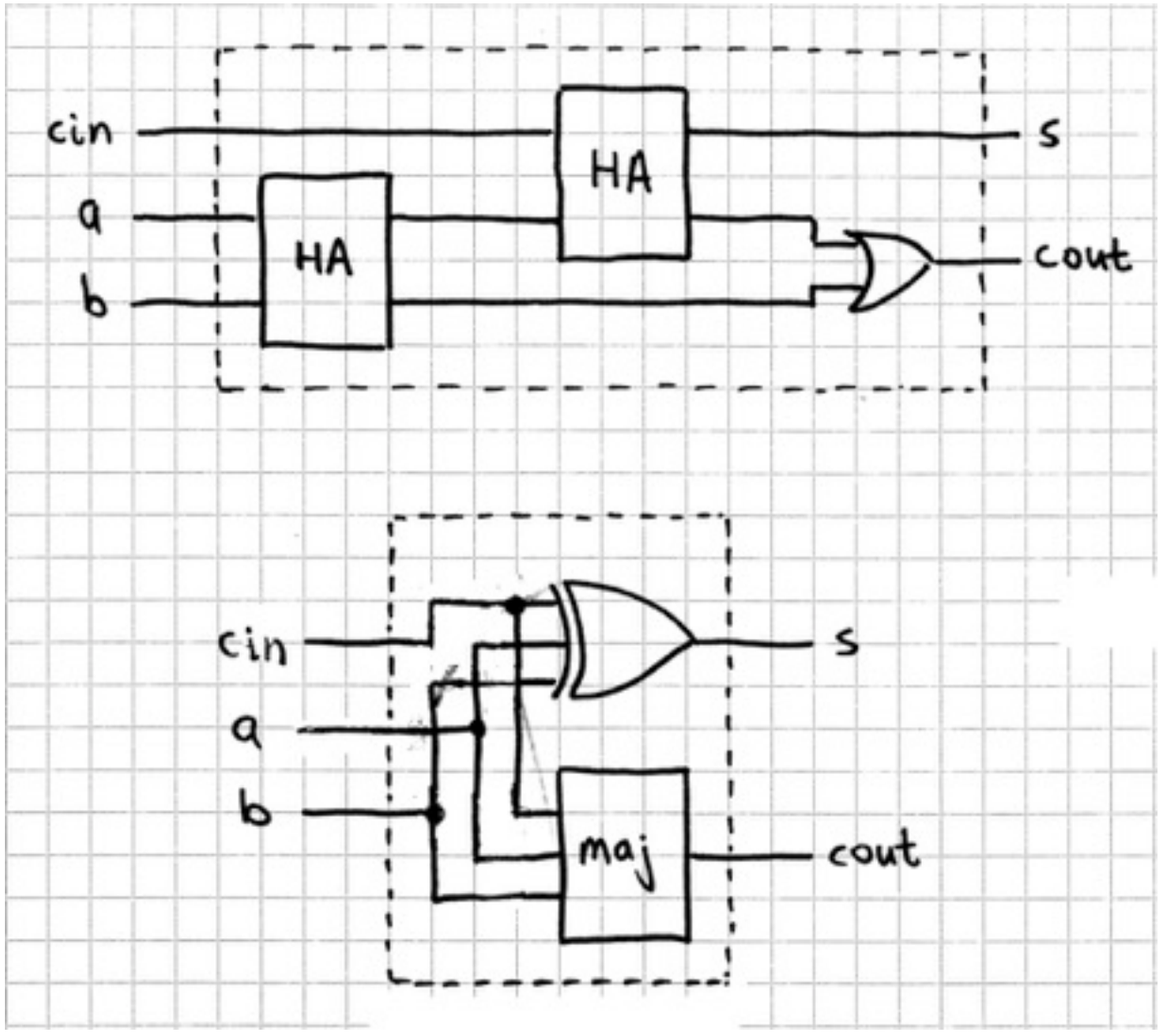


A half adder adds two binary digits to obtain a two-bit binary result.

Full adder

A full adder adds three binary digits and also produces a two-bit result.

Tradition dictates that we think of a full adder as made from two half adders.

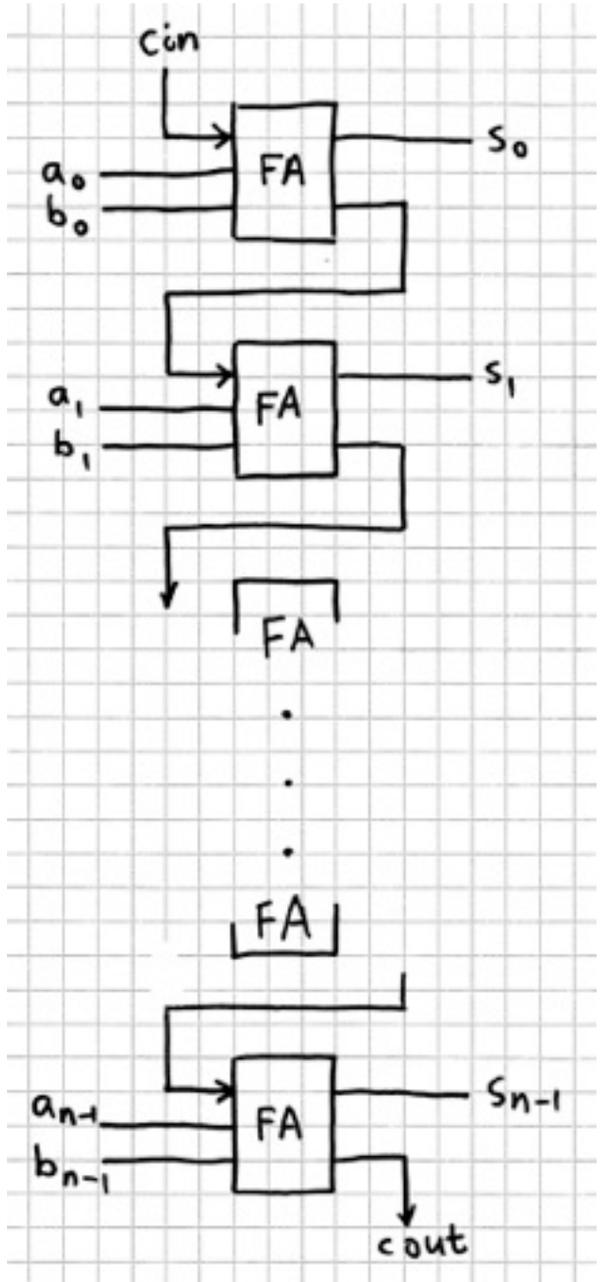


Ripple carry adder

We can add multi-bit numbers with a row of full adders.

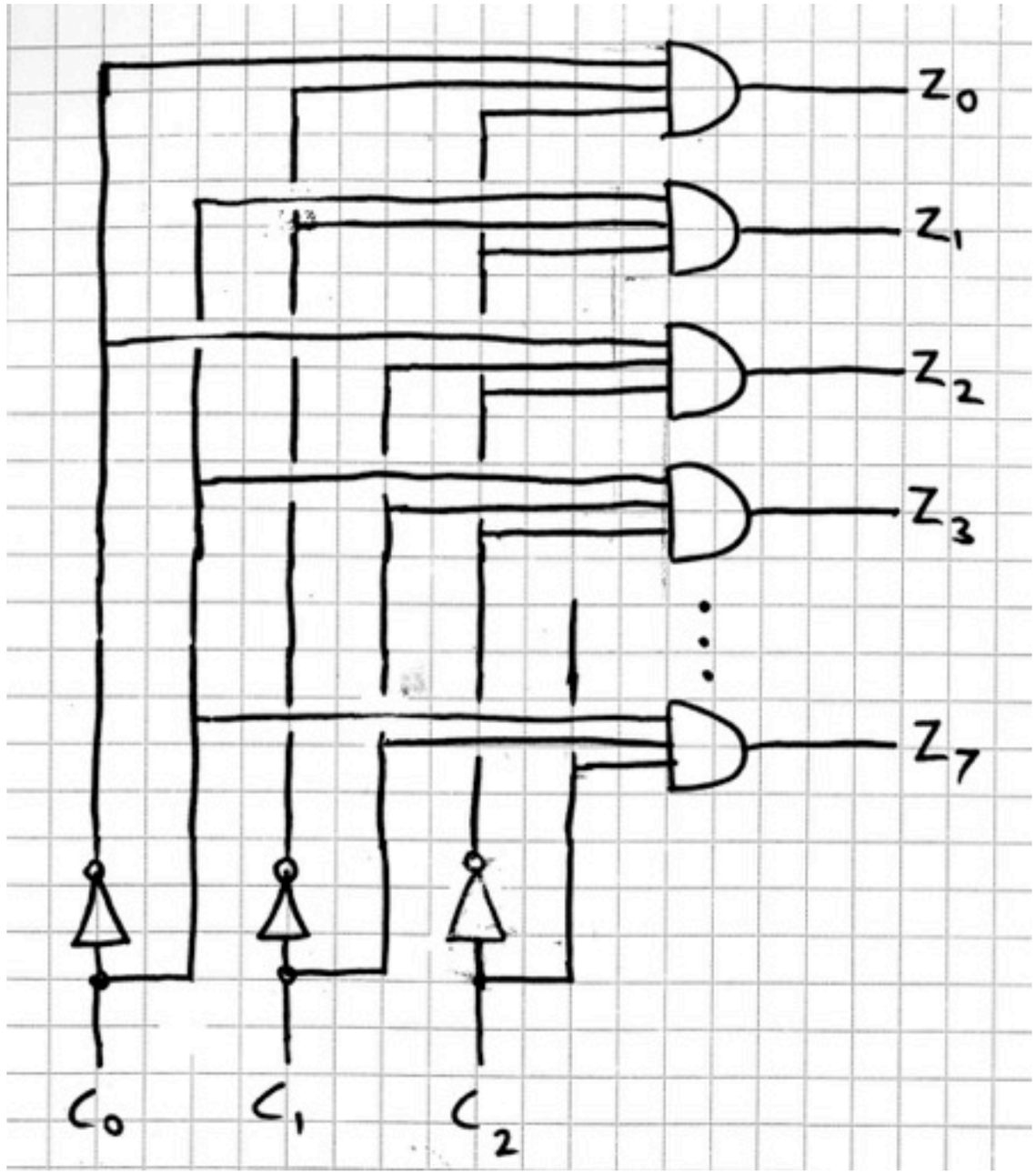
This is a bad design because of the long combinational path taken by the carries.

Better designs exist, such as the *carry-lookahead adder* with a combinational depth $O(\log n)$.

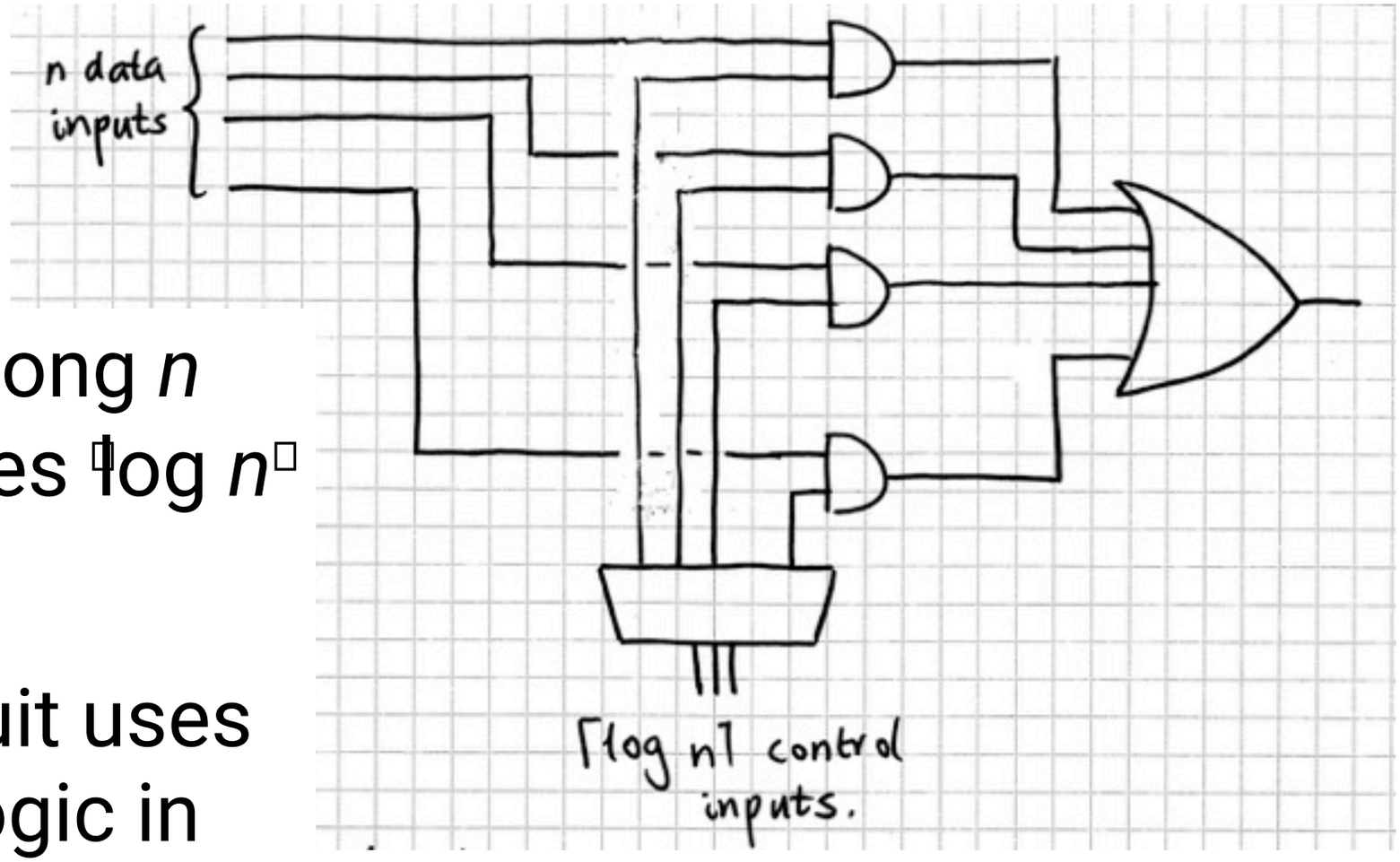


Decoder

A *decoder* has *n* inputs and 2^n outputs, one for each binary value of the inputs.



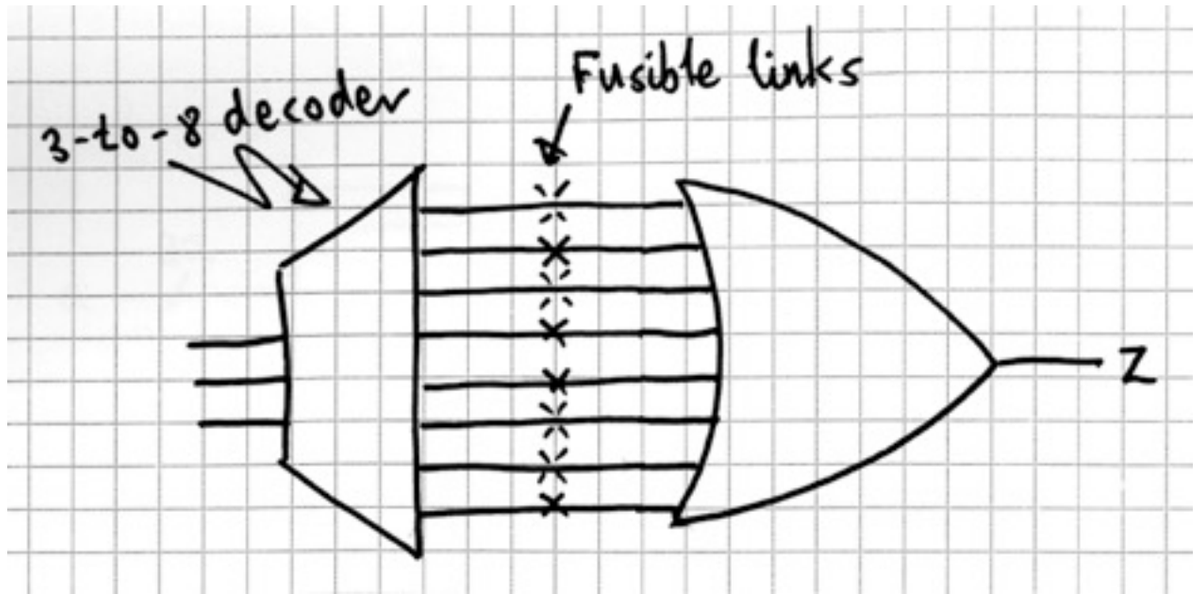
n-way multiplexer



Selecting among *n* inputs requires ⌈log *n*⌉ control bits.

A better circuit uses three-state logic in place of the AND gates and wide OR gate.

8×1-bit ROM



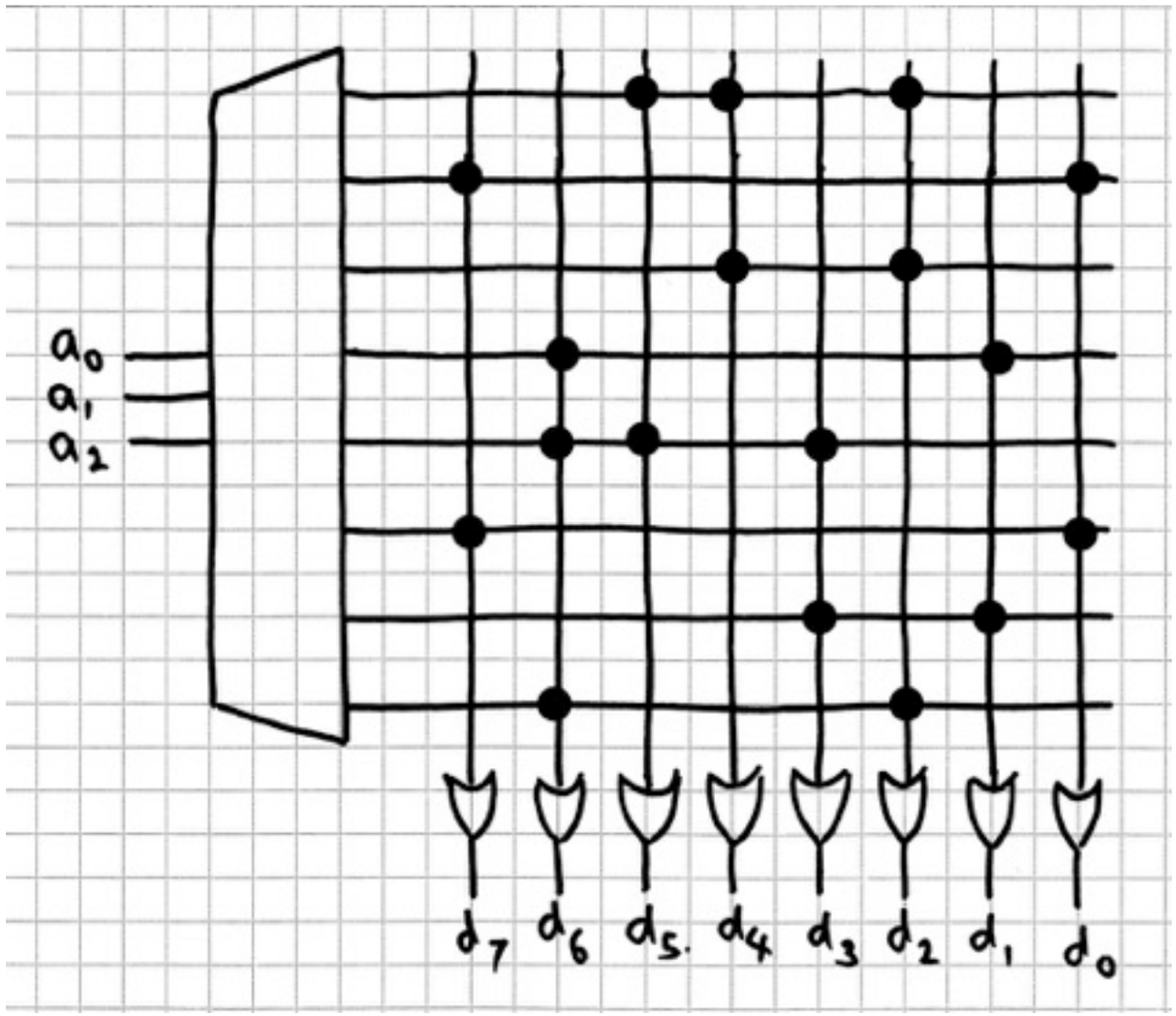
A ROM can be built from a decoder, a pattern of connections, and a wide or gate.

We can program the ROM by making or breaking the connections.

8×8-bit ROM

A wider ROM needs a decoder plus effectively a wide OR gate for each bit.

A workable design has just one transistor at each crossing.



General sequential circuit

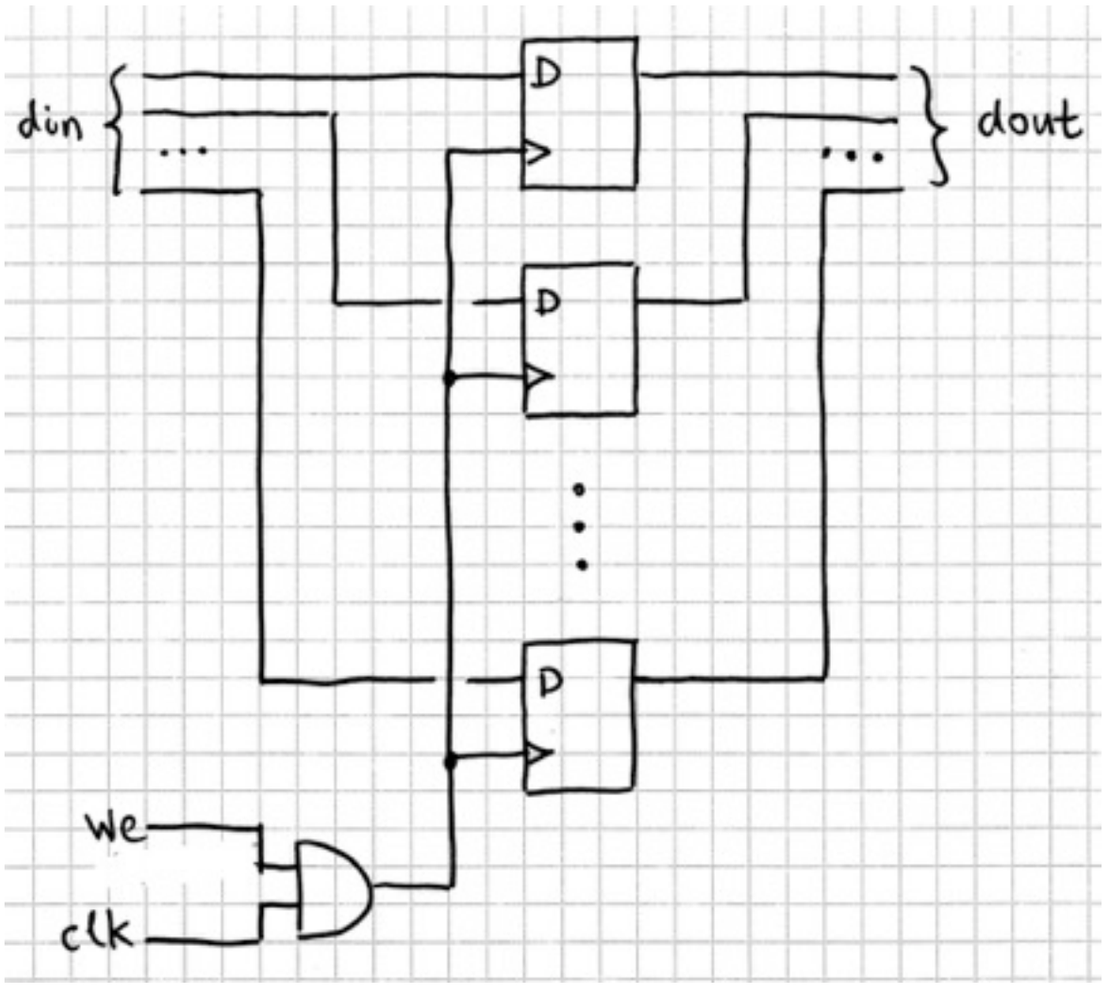
If all the state bits are placed in a row,

Then the next-state and output logic can be implemented by a ROM or something like it.

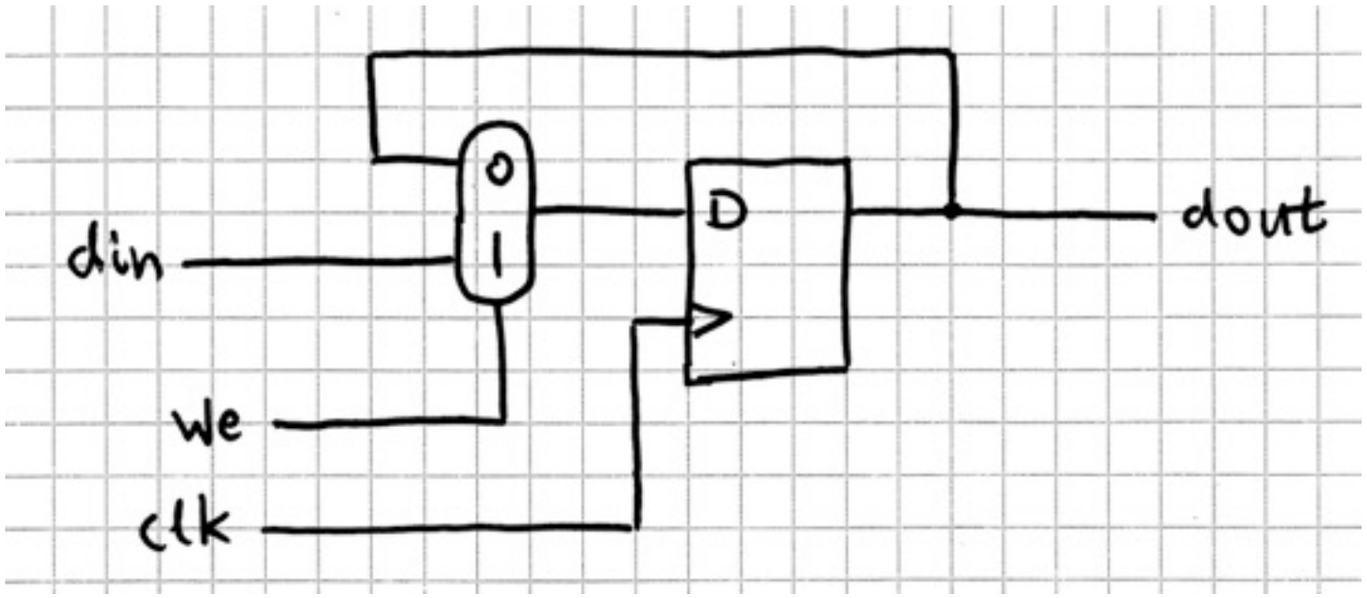
Register with write-enable

Contents are updated only if *we* is 1.

```
forever {
    dout = state;
    pause;
    if (we)
        state = din;
}
```

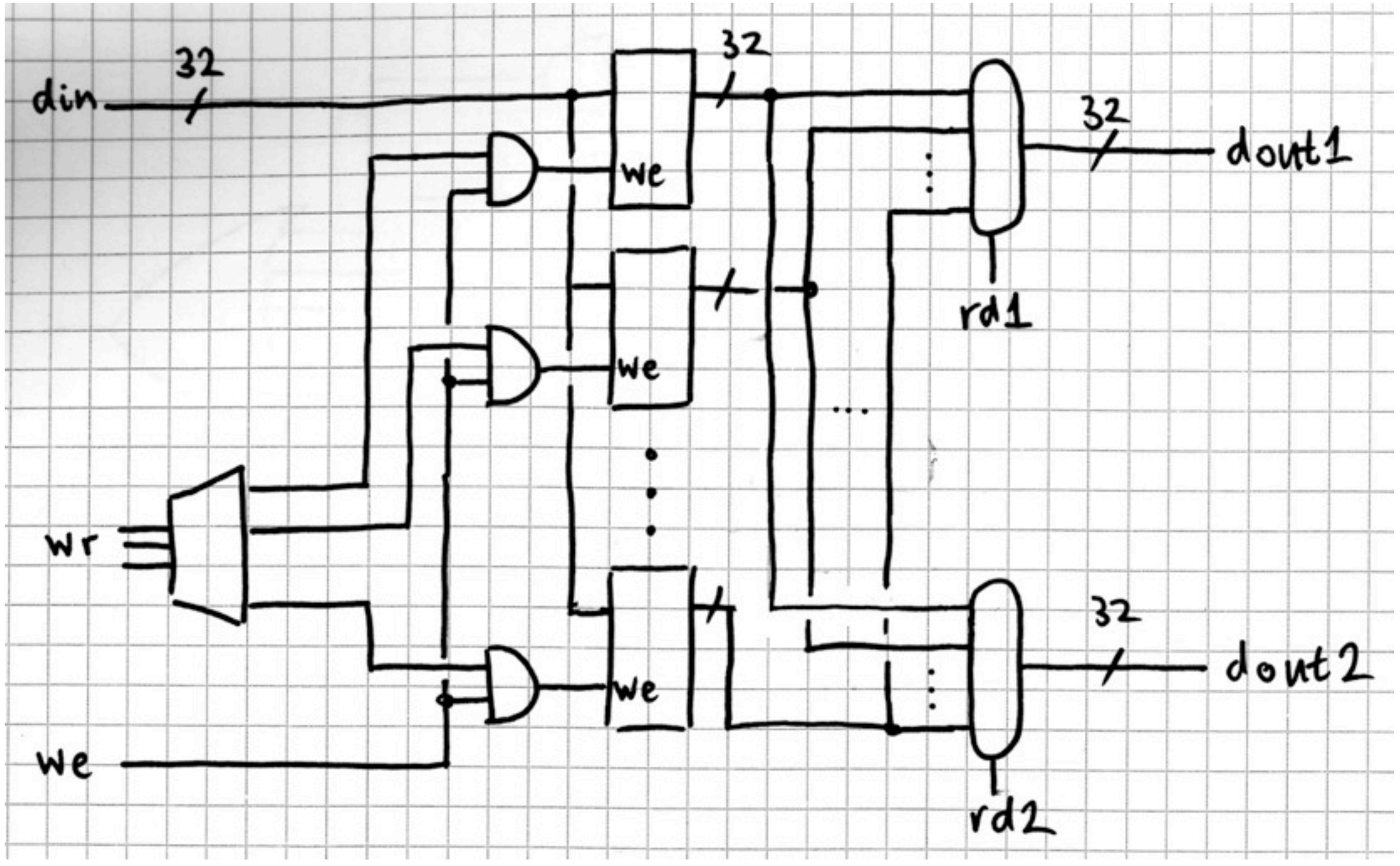


A fully synchronous alternative

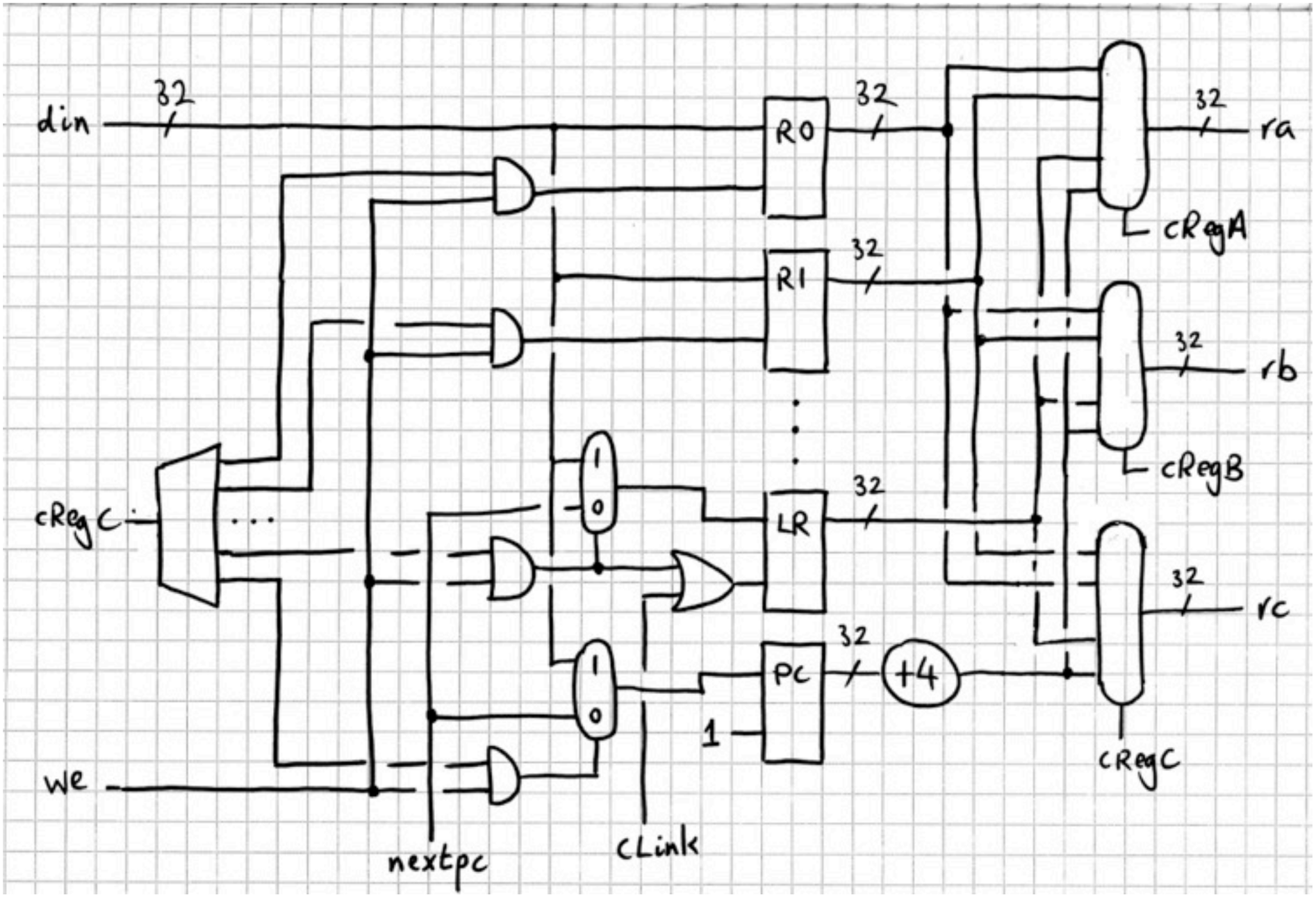



```
forever {
   dout = state;
   pause;
   state = (we ? din : state);
}
```

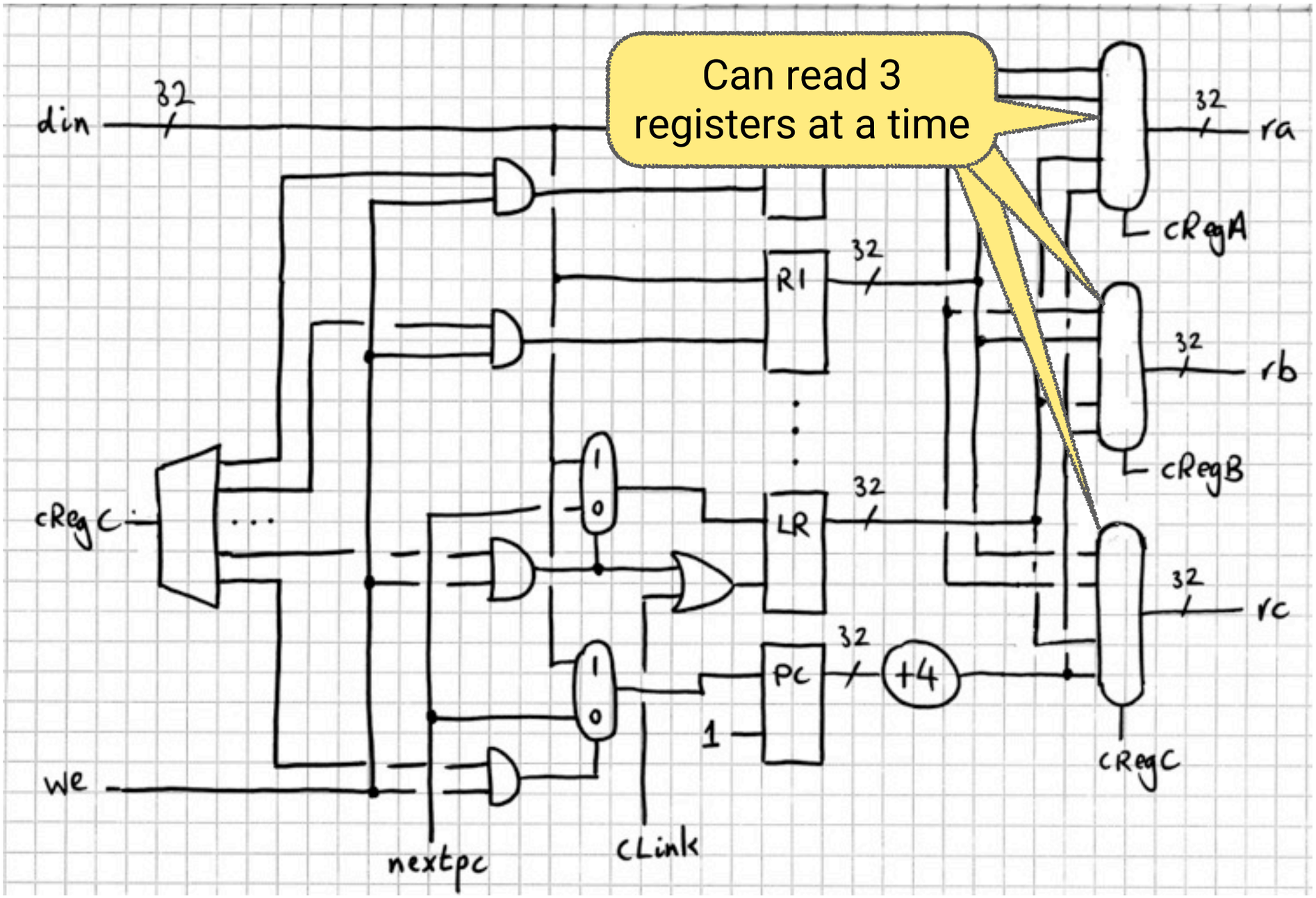
Dual-port register file



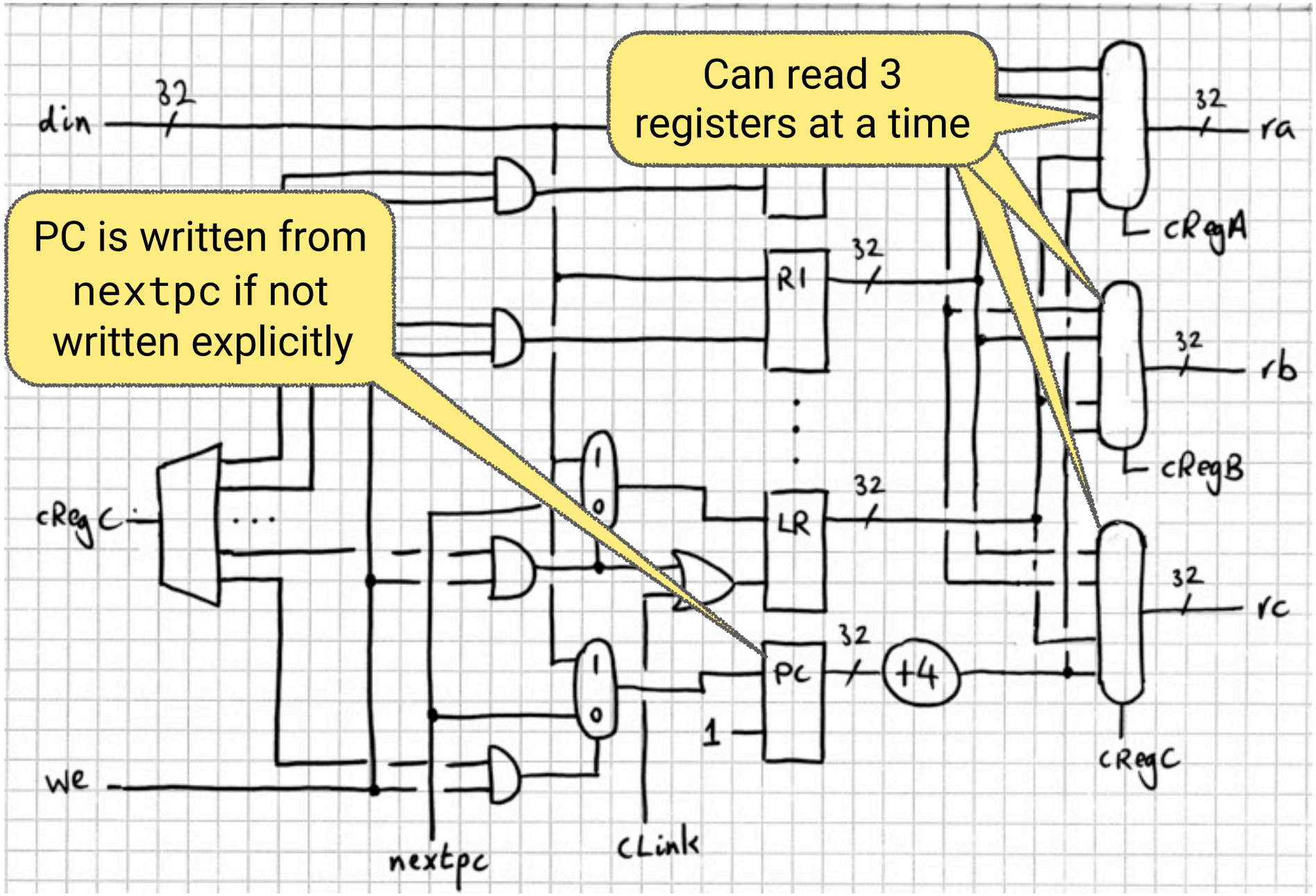
Turbo-charged register file



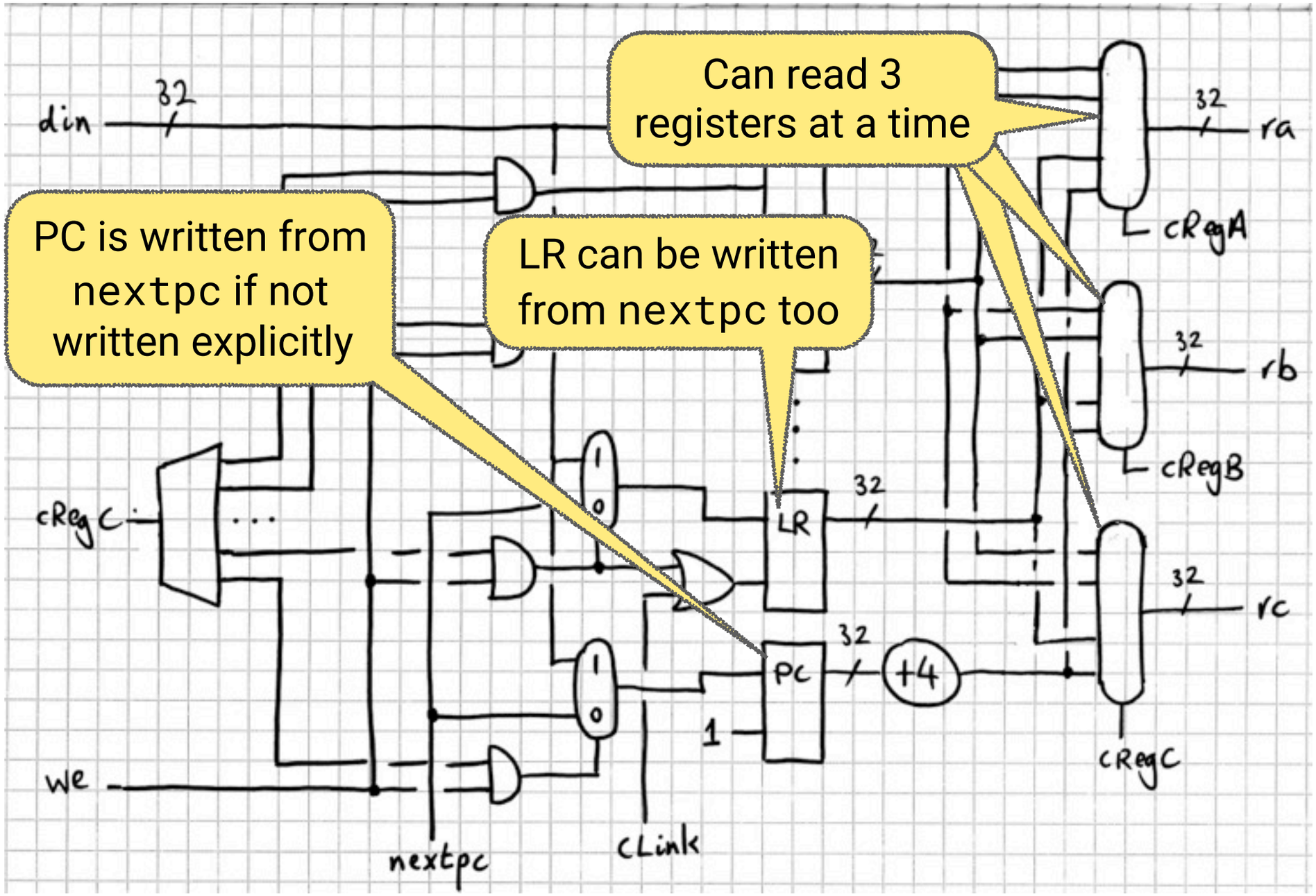
Turbo-charged register file



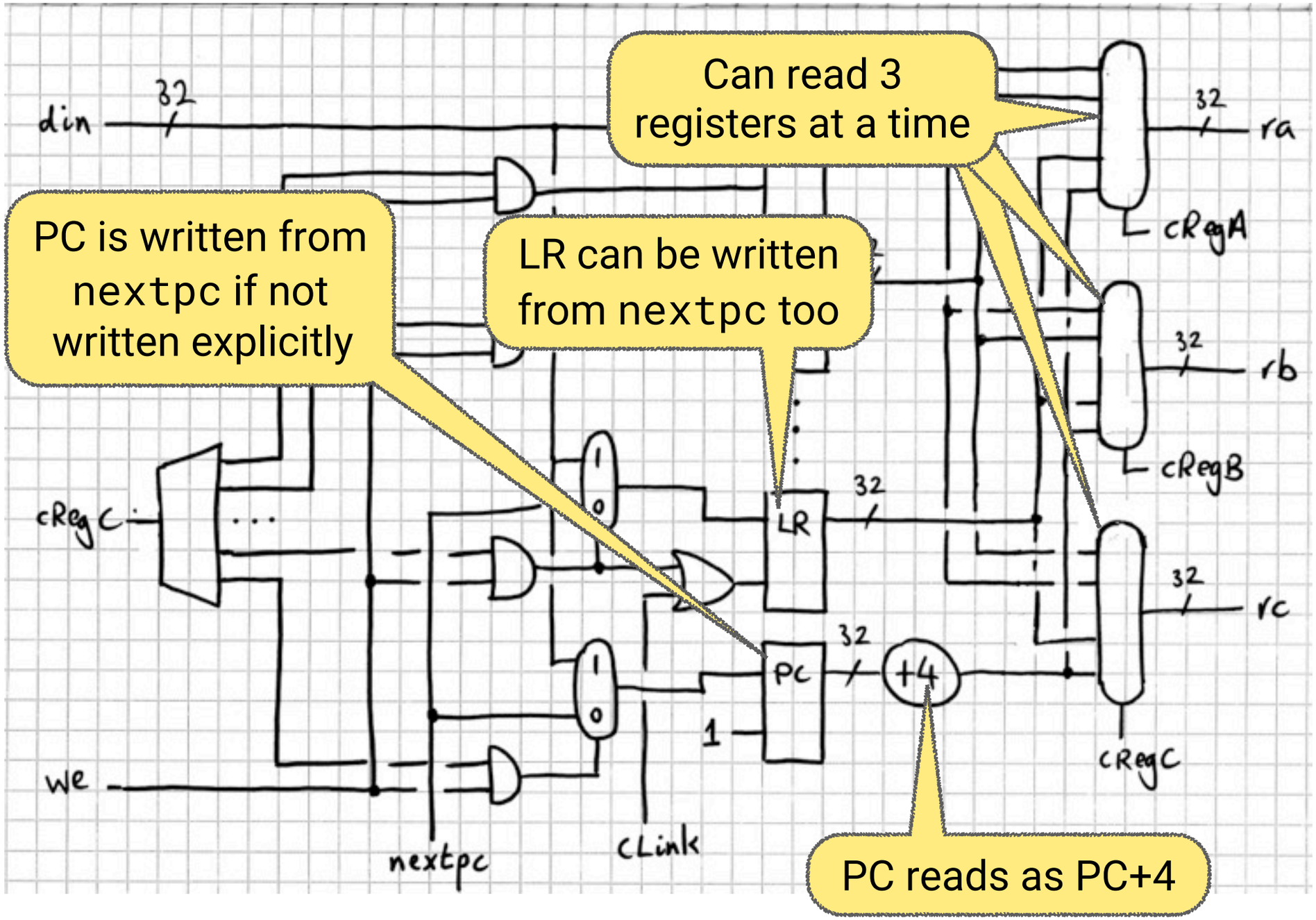
Turbo-charged register file



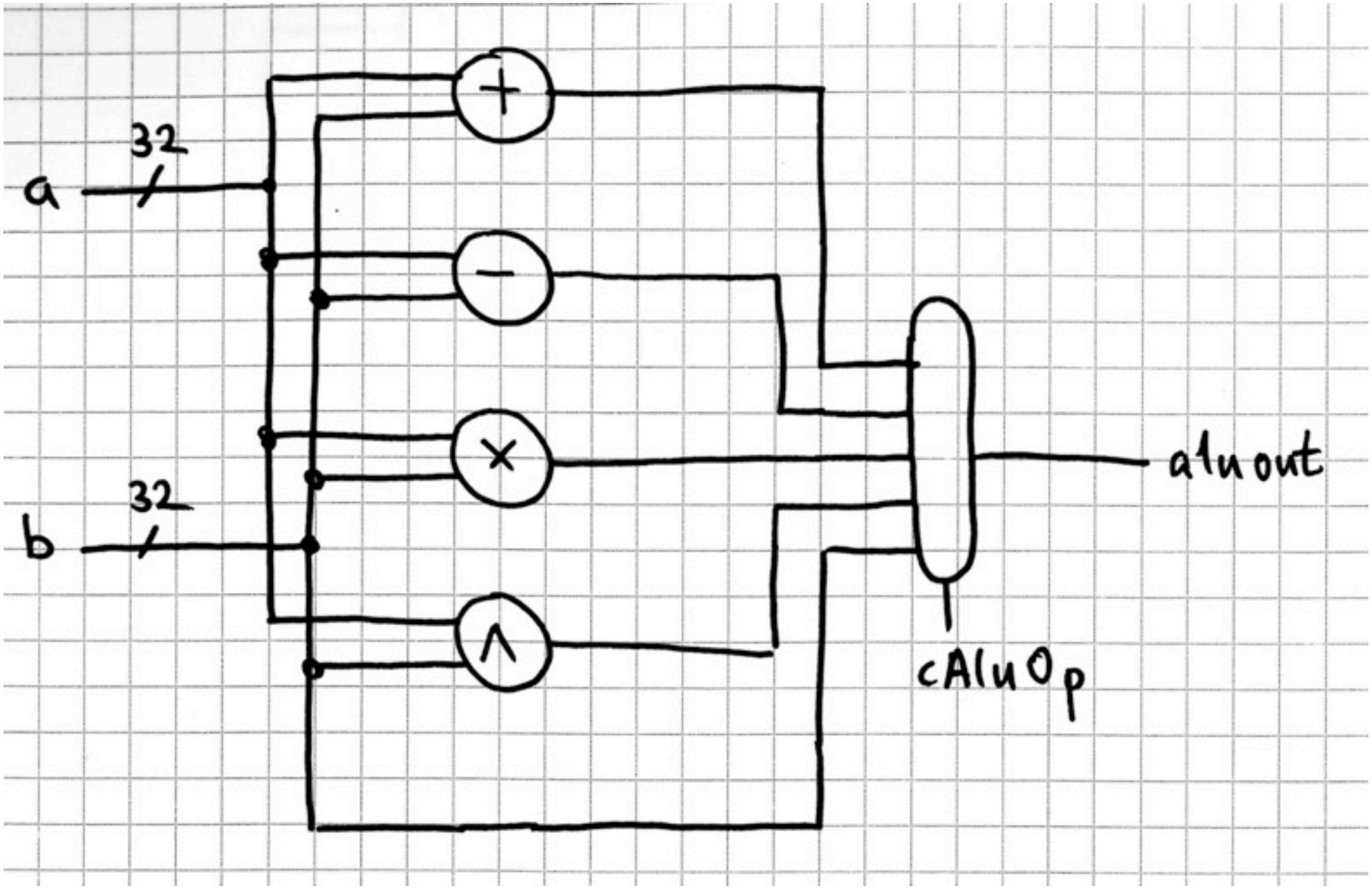
Turbo-charged register file



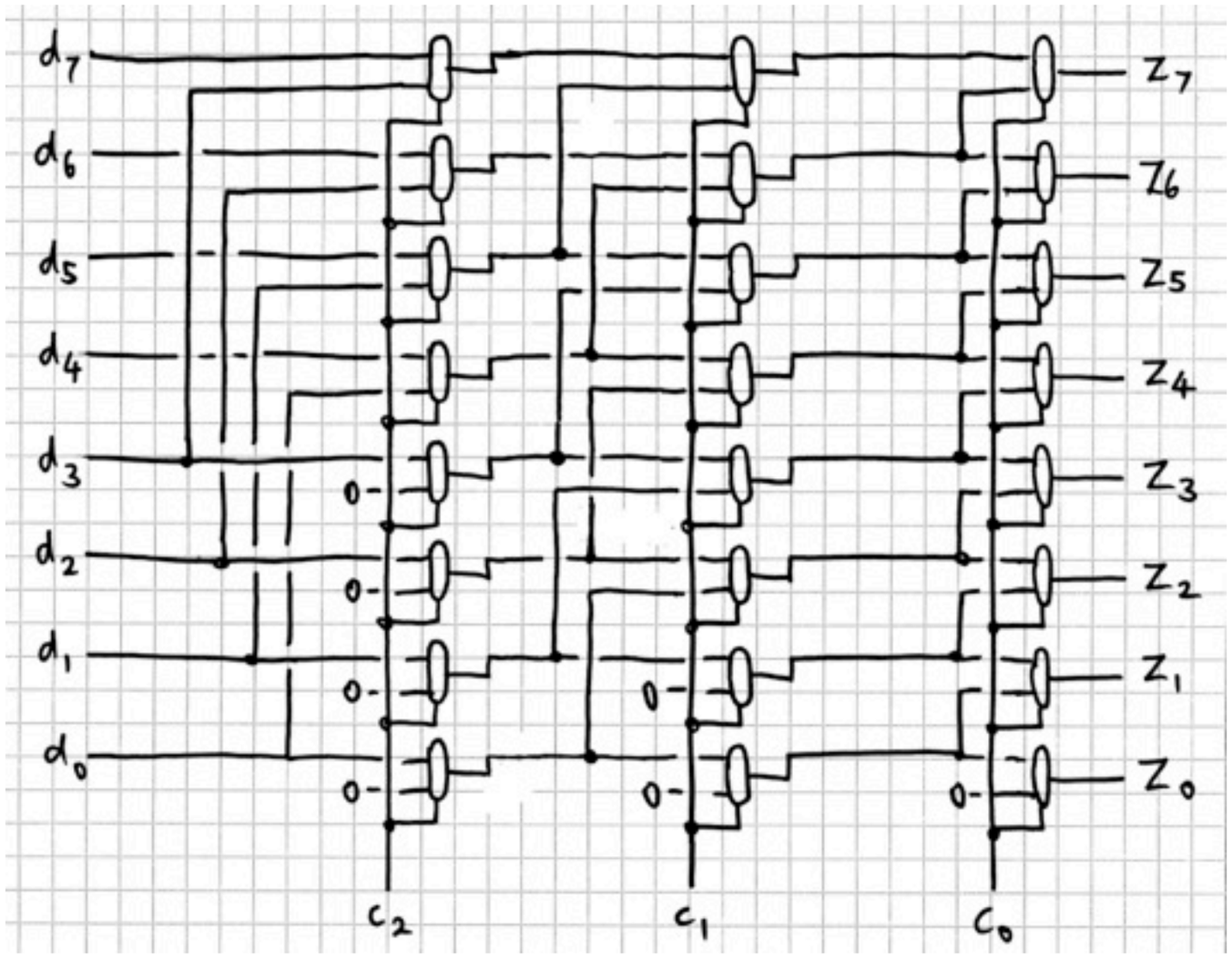
Turbo-charged register file



A simple-minded ALU



Barrel shifter



This uses the principle that $x << 5 = (x << 4) << 1$.